Classic Shell Scripting



Shell 脚本学习指南

O'REILLY®

机械工业出版社
China Machine Press

HZ BOOKS

Arnold Robbins & Nelson H. F. Beebe 著
*O'Reilly Taiwan*公司 编译

Shell脚本学习指南

Arnold Robbins & Nelson H. F. Beebe 著

O'Reilly Taiwan 公司 编译

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Taipei • Tokyo

O'Reilly Media, Inc. 授权机械工业出版社出版

机械工业出版社

图书在版编目（CIP）数据

Shell脚本学习指南 /（美）罗宾（Robbins, A.）（美）比博（Beebe, N. H. F.）著；O'Reilly Taiwan公司编译．－北京：机械工业出版社，2009.4

书名原文：Classic Shell Scripting

ISBN 978-7-111-25504-8

I. S… II. ①罗… ②比… ③ O… III. UNIX操作系统－程序设计 IV. TP316.81

中国版本图书馆CIP数据核字（2008）第170846号

北京市版权局著作权合同登记

图字：01-2008-4830号

本书法律顾问
北京市展达律师事务所

书　　名 / Shell脚本学习指南
书　　号 / ISBN 978-7-111-25504-8
责任编辑 / 曾珊
封面设计 / Emma Colby，张健
出版发行 / 机械工业出版社
地　　址 / 北京市西城区百万庄大街22号（邮政编码 100037）
经　　销 / 新华书店北京发行所发行
印　　刷 / 北京京师印务有限公司
开　　本 / 178毫米 × 233毫米　16开本　32印张
版　　次 / 2009年4月第1版　2009年4月第1次印刷
定　　价 / 79.00元（册）

凡购本书，如有倒页、脱页、缺页，由本社发行部调换
本社购书热线：(010)68326294

O’Reilly Media, Inc. 介绍

为了满足读者对网络和软件技术知识的迫切需求，世界著名计算机图书出版机构 O’Reilly Media, Inc. 授权机械工业出版社，翻译出版一批该公司久负盛名的英文经典技术专著。

O’Reilly Media, Inc. 是世界上在 UNIX、X、Internet 和其他开放系统图书领域具有领导地位的出版公司，同时也是联机出版的先锋。

从最畅销的《The Whole Internet User’s Guide & Catalog》（被纽约公共图书馆评为二十世纪最重要的 50 本书之一）到 GNN（最早的 Internet 门户和商业网站），再到 WebSite（第一个桌面PC的Web服务器软件），O’Reilly Media, Inc. 一直处于Internet 发展的最前沿。

许多书店的反馈表明，O’Reilly Media, Inc. 是最稳定的计算机图书出版商 —— 每一本书都一版再版。与大多数计算机图书出版商相比，O’Reilly Media, Inc. 具有深厚的计算机专业背景，这使得 O’Reilly Media, Inc. 形成了一个非常不同于其他出版商的出版方针。O’Reilly Media, Inc. 所有的编辑人员以前都是程序员，或者是顶尖级的技术专家。O’Reilly Media, Inc. 还有许多固定的作者群体 —— 他们本身是相关领域的技术专家、咨询专家，而现在编写著作，O’Reilly Media, Inc. 依靠他们及时地推出图书。因为 O’Reilly Media, Inc. 紧密地与计算机业界联系着，所以 O’Reilly Media, Inc. 知道市场上真正需要什么图书。

计算机精品学习资料大放送

软考官方指定教材及同步辅导书下载 | 软考历年真是解析与答案

软考视频 | 考试机构 | 考试时间安排

Java一览无余：**Java**视频教程 | **Java SE** | **Java EE**

.Net技术精品资料下载汇总：**ASP.NET**篇

.Net技术精品资料下载汇总：**C#**语言篇

.Net技术精品资料下载汇总：**VB.NET**篇

撼世出击：**C/C++**编程语言学习资料尽收眼底 电子书**+**视频教程

Visual C++(VC/MFC)学习电子书及开发工具下载

Perl/CGI脚本语言编程学习资源下载地址大全

Python语言编程学习资料**(**电子书**+**视频教程**)**下载汇总

最新最全**Ruby**、**Ruby on Rails**精品电子书等学习资料下载

数据库管理系统**(DBMS)**精品学习资源汇总：**MySQL**篇 | **SQL Server**篇 | **Oracle**篇

平面设计优秀资源学习下载 | **Flash**优秀资源学习下载 | **3D**动画优秀资源学习下载

最强**HTML/xHTML**、**CSS**精品学习资料下载汇总

最新**JavaScript**、**Ajax**典藏级学习资料下载分类汇总

网络最强**PHP**开发工具**+**电子书**+**视频教程等资料下载汇总

UML学习电子资下载汇总 软件设计与开发人员必备

经典**LinuxCBT**视频教程系列 **Linux**快速学习视频教程一帖通

天罗地网：精品**Linux**学习资料大收集**(**电子书**+**视频教程**) Linux**参考资源大系

Linux系统管理员必备参考资料下载汇总

Linux shell、内核及系统编程精品资料下载汇总

UNIX操作系统精品学习资料<电子书**+**视频>分类总汇

FreeBSD/OpenBSD/NetBSD精品学习资源索引 含书籍**+**视频

Solaris/OpenSolaris电子书、视频等精华资料下载索引

目录

计算机精品学习资料大放送

序

真的！我已经 30 年未写 Shell 脚本了？！？现在仔细想想，我想应该有吧，虽然一开始只是作些简单的工作（早期的 UNIX Shell，在 Bourne Shell 之前，是极为原始的，因此要写个实用的脚本是很难的事，幸好那段日子并不长）。

近几年来，Shell 一直被忽略，是一个不受重视的脚本语言。Shell 虽然是 UNIX 的第一个脚本语言，但它仍是相当优秀的。它结合了延展性与效率，持续保有独具的特色，并不断地被改良，使它们多年来一直能与那些花招很多的脚本语言保持抗衡。GUI 是比命令行 Shell 更流行的用户界面，但脚本语言时常都是这些花哨的屏幕图形界面最强有力的支柱，并一直称职地扮演这个角色。

Shell 需依赖其他程序才能完成大部分的工作，这或许是它的缺陷，但它不容置疑的长处是：简洁的脚本语言标记方式，而且比 C（还有其他语言）所编写的程序执行更快、更有效率。它使用通用的、一般用途的数据表示方式，文本行，在一个大的（且可扩展的）工具集中，让脚本语言能够搭配工具程序，产生无穷的组合。用户可以得到比那些独占性软件包更灵活、功能更强大的工具。Shell 的早期成功即以此法强化 UNIX 的开发哲学，构建一套专门性、单一目的工具，并将它们整合在一起做更多的事。该原则接着鼓励了 Shell 的改良，允许用这种方式完成更多的工作。

Shell 脚本还有一个超越 C 程序的优势，同样也优于其他脚本语言的地方，可用一般方式轻松地读取与修改。即便不是C的程序设计人员，也能像现今许多系统管理人员一样，很快就能接受 Shell 脚本。如此种种，让 Shell 脚本成为延展用户环境与定制化软件包的重要一环。

的确，它其实有一种“周而复始”的特性，在我看过这么多软件项目之后。项目将简单的 Shell 脚本置于关键位置，让用户容易地从他们的角度来定制软件。然而，也因为这

些项目的 Shell 脚本与周围的 C 程序码相比较，要更容易解决问题，所以不断产生更复杂的脚本。最后，它们终于复杂到让用户很难轻易地处理（我们在 C News 项目里的部分脚本就拥有著名的 Shell 压力测试，完全未考虑用户的立场），且必须提供新的脚本集合，供用户进行定制……

长久以来，一直都没有编写 Shell 脚本相关的好书出现。UNIX 程序设计环境方面的书籍偶有触及这方面议题，但通常只是简短带过，作为它众多主题的一部分，有些写得不错的书也很久没有更新了。好的参考文件应该是针对各种不同的 Shell 讨论，但必须是贴近新手的实战手册，涵盖工具程序与 Shell，以循序渐近的方式介绍，告诉我们如何得到更好的结果与输出，还要注意到实例面，像是可读性议题。最好它还讨论各式 Shell 的异同，而不是好像世上只有一个 Shell 存在一样。

这本书就是这样的，甚至做到比上面说的还多。至少，它是第一本且最好的一本、内容最新的、以最轻松的方式介绍 UNIX 脚本语言的书。以实用的范例进行解说，让工具充分发挥自己的效能。它包括了标准 UNIX 工具，让用户有个好的开始（对于觉得看手册页有点难的用户来说，这会是个相当不错的参考教材）。我最高兴的是看到将 awk 列入取材范围，这是相当有用且不容忽视的工具程序，适于整合其他工具及简洁地完成小型程序设计的工作。

我建议所有正在编写 Shell 脚本或管理 UNIX 系统的人都要读这本书。我在这本书上学到很多，我想你也会。

—— Henry Spencer
SP Systems

前言

刚开始使用UNIX（注1）的用户与程序员突然面对各式各样的程序时，都会有很多疑问，例如“它们的功能是什么”，还有“我怎么使用它们”。

本书可以回答你这些问题。告诉你如何结合UNIX工具，将其与标准的Shell相结合完成工作。Shell脚本的编写是门艺术，需要的不只是Shell语言的相关知识，还要你对各个独立的UNIX程序有基本认识：为什么会有这些工具，要怎么单纯地使用它们，怎么将它们与其他程序结合应用。

为什么需要学习如何编写Shell命令？因为大部分情况下，中型到大型的问题都能拆成较小的部分，这些小部分也多半都能找到现成的UNIX工具处理。用心编写的好用Shell脚本常常能够比C或C++语言编写的程序更快地解决相同的问题。也可以让Shell脚本提供可移植性，也就是说，可以跨越UNIX与POSIX兼容的系统，有时仅需略作修改，甚至不必修改，即可使用。

谈到UNIX程序时，我们使用工具（tool）这个字。以UNIX工具箱（toolbox）的做法解决问题，长久以来以“软件工具（Software Tools）”哲学（注2）为人所熟知。

瑞士军刀是很多人口袋里的好帮手。它有刀刃、螺丝起子、开罐器、牙签等工具。功能更齐备的，还有其他像拔塞钻、放大镜等工具。瑞士军刀能派上用场的时候很多，虽然用它来修削和进行简单雕刻很不错，但你绝不会拿它来盖狗屋或制作鸟类喂食器。相反，

注1：整本书里，我们都使用UNIX这个字，指的不单单是商用的UNIX系统原始版本，像Solaris、Mac OS X，与HP-UX，还包括可自由取得的类似系统，例如GNU/Linux与各种BSD系统：BSD/OS、NetBSD、FreeBSD，与OpenBSD。

注2：此法因《Software Tools》（Addison-Wesley）这本书而广受欢迎。

做这类工作时你会寻求更专门的工具，例如铁槌、锯子、夹钳或刨刀等。同理，当你在解决程序化问题时，使用专门的软件工具会比较好。

这是给谁看的书

这本书是写给那些在UNIX环境下发现必须写些Shell脚本，以利于工作进行的计算机用户与软件开发人员。例如，你可能是正在念计算科学的学生，手上有学校给你的第一个UNIX系统账号，你想知道在UNIX下更多的东西，例如你的Windows个人计算机无法处理的那些工作（这种情况下，你通常得写几个脚本来定制个人环境）。或者，你可能是个系统管理新手，需要为公司或学校写几个专用程序（可能是处理事件日志文件，账号、账单管理之类的事情）。你也可能是Mac OS的开发老手，但转到崭新的Mac OS X的世界，它的安装程序是以Shell脚本写成。不管你来自哪里，如果你想学Shell脚本，这本书就是写给你的。在这本书里你能学到：

软件工具设计概念与原则

一些好的软件工具设计与实例上的实践规则。我们会解释这些原则，还会在这本书里贯彻执行。

UNIX 工具是什么

UNIX的核心工具组会在我们编写Shell脚本时不断地重复使用。我们会介绍Shell与正则表达式的基本概念，并在解决特定问题时展现各种核心工具的用法。除了介绍工具能做什么之外，我们还会告诉你，为什么要使这个工具，为什么它有这些特殊选项。

《Learning UNIX》这本书是在介绍UNIX系统，让你从对UNIX毫无经验成长为会基本操作的用户。《UNIX in a Nutshell》这本书则是广泛地介绍UNIX工具包，对于使用时机与特定工具用法的介绍很少。我们的目的就在弥补这两本书之间的鸿沟：如何灵活运用这些UNIX提供的工具包，让工作更顺畅，更有效率，也更从容（我们的期望）。

如何结合所有工具，完成工作

编写Shell脚本时，其实会是“整体的功能比各部分加起来的总和还强大”。Shell的使用就像整合个别工具的黏着剂，让你只要花点心思，就能得到惊人的效果。

标准工具几个常见的扩展

如果你已经是GNU/Linux或BSD系统的用户，很可能你的工具还有其他额外的、好用的功能或选项。这部分我们也会介绍。

不可或缺的非标准工具

有些程序，在大部分传统的UNIX系统里并非“标准的”，但我们又不能没有它。我们会在适当的地方介绍它们，也会提供使用时机的相关信息。

对长期使用UNIX的开发人员与管理者来说，软件工具的设计原则一直没有什么改变。因此，推广的书籍虽然还算堪用，但已经20年未更新了，甚至更久！UNIX系统在这些书写成之后，有了许多变动。因此，我们觉得是更新这些想法的时候了，我们利用这些工具的现行版本、在现行系统下展示范例。下面是我们将要强调的部分：

- 所有的呈现是以POSIX为基础。POSIX为一系列描述可移植操作系统环境的标准正式名称的缩写。POSIX标准是开发人员的挑战，他们必须兼顾程序与Shell脚本在不同厂商所提供的各种平台上的可移植性。我们将在最新的POSIX标准下展现Shell语言、各个工具程序及其选项。

  该标准的官方名称为IEEE Std. 1003.1-2001（注3），它包括数个可选用的部分，最重要的一个就是X/Open System Interface（XSI）规格。这些功能文件详细描述了UNIX系统长久以来的行为模式。我们会介绍现行标准与早期1992标准间的差异，也会提供与XSI相关的特点。要了解UNIX相关标准，*http://www.UNIX.org/*（注4）是一个很好的起点。

  Single UNIX Specification的官方网站为*http://www.UNIX.org/version3/*。该标准可在线访问，不过得先到*http://www.UNIX.org/version3/online.html*注册。

  有时，该标准会将特殊行为模式保留为未定义（unspecified）。这是为了能让厂商以扩展的方式支持该行为，例如：额外的功能与标准本身未做成文件的部分。

- 除了告诉你如何执行特定程序外，我们还会强调这些程序存在的原因，及它们能解决什么问题。了解为什么会有这样的程序，有助于你进一步了解它的使用时机与方式。

- 大部分的程序都提供了相当多的选项组合。但通常只有一小部分是日常工作用得上的。这类程序，我们会让你知道它的哪些选项较方便好用。事实上，我们无法遍及每个程序的所有选项，未提及的部分，你可以通过程序的使用手册或其他参考书籍，例如《UNIX in a Nutshell》（O'Reilly）与《Linux in a NutShell》（O'Reilly），来了解它。

在你看完这本书之后，你不仅能了解UNIX工具集，还能吸收到UNIX的中心思想与软件工具设计的原则。

注3：该标准的2004版在本书内文底定后才发表。对学习Shell脚本而言，2001与2004间的差异并不重要。

注4：有关IEEE Std. 1003.1-2001的常见技术性问答（FAQ）文件，你或许可以在*http://www.opengroup.org/austin/papers/posix_faq.html*找到。与标准有关的后台知识，则在*http://www.opengroup.org/austin/papers/backgrounder.html*中。

你应该先有什么基础

我们认为读者应该了解以下背景知识：

- 如何登录 UNIX 系统
- 如何在命令行上执行程序
- 如何做一个简单的命令管道，与使用简单的输出/入重定向，例如 < 与 >
- 如何以 & 将程序放到后台执行
- 如何建立与编辑文件
- 如何使用 chmod，将脚本设为可执行权限

再者，如果你想试着操作本书范例，在你的终端机（或终端机模拟器）下达命令时，我们建议你使用POSIX 兼容的 Shell，例如 ksh93 的最新版本，或 bash 的近期版本。请特别注意：商用 UNIX 系统上的 /bin/sh 可能并非完全兼容于 POSIX。

ksh93、bash 与 zsh 的下载网址请见第 14 章。

各章介绍

我们建议你依序阅读本书，因为每个章节都与前面章节息息相关。我们在此逐章介绍如下。

第 1 章，背景知识
: 此处提供简短的 UNIX 历史沿革。特别是贝尔实验室的运算环境，也就是 UNIX 开发的地方，激发了许多软件工具设计上的哲学。该章还会介绍这些原则，并在本书中贯彻执行。

第 2 章，入门
: 该章从编译语言与脚本语言间的取舍开始讨论。之后再介绍两个相当简单但很实用的Shell脚本程序。涵盖范围包括了命令、选项、参数、Shell变量、echo与printf的输出、基本输入/输出重定向、命令查找、从脚本里访问参数以及执行跟踪。最后则以国际化与本地化结束；这是在今日“地球村”环境下渐受重视的议题。

第 3 章，查找与替换
: 这里会介绍以正则表达式进行文字查找（或比对）。我们还会说明修改与提取文字的操作。这些都是最基本的 Shell 脚本编写的操作。

第4章，文本处理工具

该章介绍的是一些文字处理的软件工具，这些在Shell脚本编写时，都会一再地使用。其中最重要的两个就是sort与uniq，在重组与降低数据量上，它们扮演很重要的角色。本章还会带你看看如何重新编排段落、计算文字单位、显示文件以及取出文件的前几行、后几行数据。

第5章，管道的神奇魔力

该章以几个小型脚本为例，展示结合简单的UNIX工具程序能够产生更强大、更灵活的工具。本章的内容采取cookbook（问题描述与解决方案）的形式，它们共同的部分在于所有的解决方案都组合自线性的管道（pipelines）。

第6章，变量、判断、重复动作

这章介绍Shell语言里不可或缺的部分。包含了Shell变量与算法、退出状态的重要概念、如何判断，以及Shell循环的处理。最后以Shell的函数作结束。

第7章，输入/输出、文件与命令执行

该章为Shell描述的另一章，也是结尾，重点放在输入/输出、Shell所执行的各种替换、加引号、命令行执行顺序，以及Shell内置命令上。

第8章，产生脚本

我们在这里会示范如何结合UNIX的工具以处理更复杂的文本处理工作。本章的程序比第5章的还大，但仍是几分钟便能消化掉。甚至它们所完成的工作，如果使用传统的程序语言，例如C、C++或Java™来做，会很困难。

第9章，awk的惊人表现

该章介绍的是awk语言必备的组成部分。awk是一套功能强大且自给自足的语言。而awk程序更可用来与其他软件工具箱里的其他程序相结合，以执行简单的数据提取、处理与格式编排工作。

第10章，文件处理

该章介绍了处理文件的几个主要工具。包括列出文件、产生临时文件，以及利用指定标准寻找文件的find命令。另外还有两个与磁盘空间有关的重要命令，以及比较文件间异同的几个程序。

第11章，扩展实例：合并用户数据库

将所有东西串起来，解决既有趣又难易适中的挑战性工作。

第12章，拼写检查

该章利用拼写检查的问题，展现如何以数种方式解决它。这里展现了原始的UNIX Shell脚本管道以及两个小型的脚本：ispell与aspell命令，可自由下载，它们更适用于批处理的拼写检查工作。我们以awk写了一个大小适当的拼写检查程序，充分展现使用该语言的简单利落。

第 13 章，进程

该章将重点从文本处理的领域转到工作（job）与系统管理上。我们介绍了几个用于管理进程的必备工具，还有`sleep`命令，这在脚本需要等待某些事发生时很有用，另外则是其他一些用于延迟的标准工具，或修正日期时间命令的处理。最重要的是，该章也包括了`trap`命令，它可以让 Shell 脚本控制 UNIX 的信号。

第 14 章，Shell 可移植性议题与扩展

这里介绍的是一些更有用的扩展，可使用于`ksh`与`bash`之下，而非 POSIX。一般情况下，你都能安心地将这些扩展套用在你的脚本里。该章还会带你看几个"gotchas"，这是等待粗心大意的Shell脚本编写者跳入的陷阱。内容包括了在编写脚本时该注意的事项，还有在执行时可能出现的矛盾。除此之外，还包括有`ksh`与`bash`的下载与安装。该章最后会探讨各种不同的Shell实现间，Shell初始化与终结的差异。

第 15 章，安全的 Shell 脚本：起点

该章会粗略介绍编写 Shell 脚本时的安全性议题。

附录 A，编写手册页

该附录讲的是如何编写手册页。这个必备的技术，在传统的UNIX的书籍里常被忽略。

附录 B，文件与文件系统

这个附录会介绍UNIX的字节数据流文件系统模型，并与较复杂的文件系统对照，然后解释其简洁的好处。

附录 C，重要的 UNIX 命令

该附录提供了许多UNIX命令的列表。建议你了解这些命令，它们可以增强你的能力。

参考书目

这是 UNIX 的 Shell 脚本编写的其他资料来源。

本书惯例

我们假设你已经知道，输入 Shell 命令时，最后会按下 Enter。Enter 在某些键盘上被表示为 Return。

提到 Ctrl-X 时，X 指的是任意字母，是在你按住 Ctrl（或 Ctl，或 Control）之后，接着按下的键。虽然我们这里用的是大写，不过你在按这个字母的时候无须按住 Shift 键。

其他特殊字符有换行符号（同于 Ctrl-J）、Backspace（同于 Ctrl-H）、Esc、Tab，与 Del（有时被标示为 Delete 或 Rubout）。

本书使用下列字体惯例：

斜体（*Italic*）

用在电子邮件地址、Internet URL、使用手册的引用。还用以表示参数，表示你在使用时，可以将它置换为你要的实际值，以及为范例提供说明性文字。

等宽字体（`Constant Width`）

提及UNIX文件名、扩展与内置命令、命令选项时，我们就会用到这个字体。除此之外，变量名称、Shell关键字、选项、函数、文件名结尾、范例中显示文件内容或命令的输出，以及当命令行或输入范例存在于正规文字中时，我们也会以等宽字体表示。简而言之，任何与计算机使用相关的，我们都用这个字体。

粗体等宽字体（**`Constant Width Bold`**）

这种字体用以区分比对文字中的正则表达式与Shell通配字符样式。在范例中，显示用户与Shell间的互动，我们也会使用这个字体，所有用户应键入的，我们都以粗体等宽字体显示，像这样：

```
$ pwd                              用户输入这个
/home/tolstoy/novels/w+p           系统显示这个
$
```

斜体等宽字体（*`Constant width italic`*）

这个字体是用在范例与内文中，需置换为正确值的命令行参数上。例如：

```
$ cd directory
```

注意：表示诀窍、建议或一般注意事项。

警告：表示警告与提醒。

参照UNIX User's Manual时，我们会使用标准形式：*name*(N)，*name*为命令名称，而N为section编号（通常是1），也就是寻找信息的地方。例如*grep*(1)即`grep`的Section 1的手册页。参考文件我们使用man page，或直接简写为manpage。

UNIX系统调用与C程序库，我们都这么写：`open()`、`printf()`。你可以使用`man`命令，查看这两者的manpage：

```
$ man open             查看open(2)的manpage
$ man printf           查看printf(3)的manpage
```

当我们要介绍一个程序时，就使用下面的方式，显示在正文附近，说明该工具程序与它的重要选项、语法与用途。

范例

语法

```
whizprog [ options ... ] [ arguments ... ]
```

说明如何执行这里指出的whizprog命令。

用途

说明此程序存在的目的。

主要选项

列出此程序每天要用到的重要选项。

行为模式

概括程序所做的事。

警告

所有要注意的事全在这。

程序码范例

本书并非只是解释命令与程序的功能，还提供了完整的Shell命令与程序的设计范例，便于用户或程序员直接应用于日常工作上。我们非常鼓励你修改与强化这些范例。

本书所提供的程序代码，都公开在GNU General Public License（GPL）条款下，允许程序复制、再利用与编修。请参阅范例包括的COPYING文件，了解许可权的实际条款。

代码可从原书网站获得：*http://www.oreilly.com/catalog/shellsrptg/index.html*。

我们会感谢你在使用程序码范例时注明出处，但这并非必要。这通常包括标题、作者、出版商与ISBN。例如：《Classic Shell Scripting》，作者Arnold Robbins与Nelson H.F. Beebe。版权所有2005 O'Reilly Media, Inc.，ISBN为0-596-00595-4。

Windows系统下的UNIX工具程序

很多的程序员初次接触UNIX系统后，再回到个人计算机的世界，常会希望也有一个像UNIX那样好用的环境（特别是在面对难以处理的MS-DOS命令行时），所以会有UNIX Shell式的界面出现在一些小型计算机的操作系统上，也不是什么奇怪的事。

近几年，我们不只看过Shell仿制品，还看过整个完整的UNIX环境的仿制品。其中两个就是使用bash与ksh93，其他则是提供自有的Shell重新实现（reimplementation）。本节将依次（以字母顺序）予以介绍，并附上联络方式与Internet下载信息。

Cygwin

Cygnus Consulting（现为 Red Hat）建立了 cygwin 环境。首先出现的是提供 UNIX 系统调用模拟器的共享程序库 cgywin.dll，该公司释出了许多 GNU 工具程序，供各种不同的 Microsoft Windows 版本使用。模拟器还包括了 Berkeley socket API 的 TCP/IP 网络。在 Windows/NT、Windows 2000 与 Windows XP 下，功能性最佳，不过在 Windows 95/98/ME 下也是可以运行的。

cygwin 环境使用自己的 Shell、自己的 C 编译器 GCC，以及搭配其 UNIX 工具集里的 GNU 工具程序。设计精良的 mount 命令提供了将 Windows C:\path 的标记方式对应到 UNIX 文件名。

http://www.cygwin.com/ 是你了解 cygwin 项目的起点。第一步就是下载它的安装程序，执行时，你可以选择要安装哪些额外包。整个安装过程都是在 Internet 上进行，没有官方的 cygwin CD，至少项目的维护人员并没有提供。

DJGPP

DJGPP 程序组提供了 MS-DOS 环境下所使用的 32 位 GNU 工具程序。以下内容摘自其网页：

> DJGPP 为执行 MS-DOS 的 Intel 80386（及更高级的）个人计算机提供了完整的 32 位 C/C++ 开发系统。其中包含许多 GNU 开发工具程序。这些开发的工具必须在 80386 或更高级的计算机上执行产生程序。大部分情况下，其所产生的程序可做商业用途而无须授权或版税。

其名称一开始是来自 D.J. Delorie，D.J. Delorie 将 GNU C++ 编译器 g++ 移植到 MS-DOS，而 g++ 一开始的名称为 GPP。之后逐渐茁壮成长，成为 MS-DOS 下完整的 UNIX 环境不可或缺的要素，并带有 GNU 工具，以 bash 作为其 Shell。不同于 cygwin 或 UWIN 的是：不需要使用任何的 Windows 版本，只要有完整的 32 位处理器与 MS-DOS 即可（当然，你也可以在 Windows 的 MS-DOS 窗口下使用 DJGPP）。官方网站为 *http://www.delorie.com/djgpp/*。

MKS Toolkit

个人计算机世界里的 UNIX 环境有许多都是以 Mortice Kern Systems 的 MKS Toolkit 建立的：

MKS Canada – Corporate Headquarters
410 Albert Street
Waterloo, ON
Canada N2L 3V3
1-519-884-2251
1-519-884-8861 (FAX)
1-800-265-2797 (Sales)
http://www.mks.com/

MKS Toolkit版本非常多，根据你的开发环境和开发人员的数量来决定要用哪一个版本。其中包含了与POSIX兼容的Shell，拥有1988 Korn Shell里的所有功能，超过300种的工具组，例如awk、perl、vi、make等。MKS程序库支持超过1500个UNIX API，使它更为完整，更易于应用在Windows环境下。

AT&T UWIN

UWIN包是David Korn与同事为了在Microsoft Windows下使用UNIX环境而产生的项目。其架构类似先前讨论的cygwin。共享程序库posix.dll提供了UNIX系统调用API的模拟器。其系统调用模拟器相当完整。其中一个有趣的创新就是将Windows的登录改为可在/reg文件系统下访问的方式。UWIN环境依赖原始的Microsoft Visual C/C++编译器，不过仍可自行下载GNU开发工具用于UWIN下。

*http://www.research.att.com/sw/tools/uwin/*是此项目的网页，说明可供下载的有哪些，并附上二进制文件的下载链接，还有与UWIN的使用许可权相关的信息。除此之外，另有UWIN的各类报告、额外的好用软件及其他类似包的链接。

UWIN包最大的优势为：它的Shell是一个真正的ksh93，因此在与UNIX的ksh93版本的兼容性上不会有问题。

联系我们

请将关于本书的意见和问题通过以下地址提供给出版商：

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

100035 北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室
奥莱利技术咨询（北京）有限公司

O'Reily公司是世界性的计算机信息出版公司。我们永远乐意听到读者对出版物的意见，包括如何让本书可以更好的建议，指正本书的错误或是读者建议本书往后改版时应该再加入的其他主题。以下是英文原书的联络数据：

http://oreilly.com/catalog/9780596005955/index.html

如果想要发表关于本书的评论和技术问题，请发邮件至：

bookquestions@oreilly.com
info@mail.oreilly.com.cn

关于图书、会议、资源中心和 O'Reilly 网络的更多信息，请查看我们的站点：

http://www.oreilly.com
http://www.oreilly.com.cn

致谢

我们俩要感谢对方的付出。虽然我们未曾见面，但协同作业的工作一直进行得很好。除了对彼此献上最热忱的感激，还要谢谢我们挚爱的妻子，谢谢她们在这段期间的贡献、耐心、爱，以及在此书编写期间的支持。

bash的维护人员 Chet Ramey 回答了许许多多与 POSIX Shell 相关的细微问题。AT&T Research 的 Glenn Fowler 与 David Korn，与 GNU Project 的 Jim Meyering，也提供不少解答。我们以字母顺序排列下面要致谢的人：Keith Bostic、George Coulouris、Mary Ann Horton、Bill Joy、Rob Pike、Hugh Redelmeier、Dennis Ritchie，他们为 UNIX 的历史问题解惑。由于 O'Reilly Media 的 Nat Torkington、Allison Randall、Tatiana Diaz 的指导，让本书由概念逐渐成型。感激 O'Reilly 的 Robert Romano 将我们原始的 ASCII 草图与 pic 略图作成图片。也谢谢 Angela Howard 为本书制作全面性索引，造福读者。

以下仍以字母顺序：谢谢 Geoff Collyer、Robert Day、Leroy Eide、John Halleck、Mark Lucking 与 Henry Spencer 为本书初稿再作技术上的检阅，及 Sean Burke 审阅第 2 版草稿。谢谢他们提供无价且非常有帮助的回馈。

UNIX Guru 的 UNIX Guru（精神导师）Henry Spencer，谢谢你为本书作的序。

感谢University of Utah的Electrical and Computer Engineering、Mathematics，与Physics等系所，以及 Center for High-Performance Computing，提供 UNIX 系统供访问；还有 IBM 与 Hewlett-Packard 提供访客访问，让我们取得编写本书所需的软件以供检测，谢谢，谢谢他们。

—— Arnold Robbins

—— Nelson H.F. Beebe

第1章

背景知识

本章将简述UNIX系统的发展史。了解UNIX在何处开发、如何开发，以及它的设计动机。这有助于用户善加利用UNIX所提供的工具。此外，本章将介绍软件工具的设计原则。

1.1 UNIX简史

或许你对于UNIX的发展史已有些了解，并且已经有很多介绍UNIX完整发展历史的资料。这里，我们只想让你知道：UNIX是在何种环境下诞生的，以及它如何影响软件工具的设计。

UNIX最初是由贝尔电话实验室（Bell Telephone Laboratories，注1）的计算机科学研究中心（Computing Sciences Research Center）开发的。第一版诞生于1970年——也就是在贝尔实验室（Bell Labs）退出Multics项目不久。在UNIX广受欢迎的功能中，有许多便是来自Multics操作系统。其中最著名的有：将设备视为文件，以及特意不将命令解释器（command interpreter）或Shell整合到操作系统中。更完整的历史信息可在 *http://www.bell-labs.com/history/unix* 找到。

由于UNIX是在面向研究的环境下开发的，因而没有必须生产或销售成品的盈利压力。这使其具有下列优势：

- 系统由用户自行开发。他们使用这套系统来解决每天所遇到的计算问题。

注1：该名称至今已变更数次。本书从这里开始都以口语式名称"贝尔实验室"(Bell Labs)称呼它。

- 研究人员可以不受拘束地进行实验，必要时也可任意变换程序。由于用户群不大，若程序有必要整个重写，多半也不会太难。由于用户即为开发人员，发现问题时便能随即修正，有地方需要加强时，也可以马上就做。

  UNIX 已历经数个版本，各个版本以字母V加上数字作为简称，如V6、V7，等等。（正式名称则是遵循发行的使用手册的修订次数编号来命名，例如 First Edition、Second Edition，等等。这两种名称的对应其实很直接：V6 = Sixth Edition、V7 = Seventh Edition。和大多数有经验的UNIX程序员一样，这两种命名方式我们都会用到）。影响最深远的UNIX系统是1979年所发行的第7版（Seventh Edition），但是在最初的几年，它仅应用于学术教育机构领域。值得一提的是：第7版的系统同时提出了 awk 与 Bourne Shell，这二者是 POSIX Shell 的基础，同时，第一本讨论 UNIX 的书也在此诞生。

- 贝尔实验室的研究人员都是计算机科学家。他们所设计的系统不单单是自己使用，还要分享给同事 —— 这些人一样也是计算机科学家。因此，衍生出“务实”（no nonsense）的设计模式：程序会执行你所赋予的任务，但不会跟你对话，也不会问一堆“你确定吗？”之类的问题。
- 除了精益求精，在设计与问题解决上，他们也不断地追求“优雅”（elegance）。关于“优雅”有一个贴切的定义：简单就是力量（power cloaked in simplicity，注2）。贝尔实验室自由的环境，所造就的不仅是一个可用的系统，也是一个优雅的系统。

当然，自由同样也带来了一些缺点。当UNIX流传至开发环境以外的地方，这些问题也逐一浮现：

- 工具程序之间存在许多不一致的地方。例如，同样的选项字母，在不同程序之间有完全不一样的定义；或是相同的工作却需要指定不同的选项字母。此外，正则表达式的语法在不同程序之间用法类似，却又不完全一致，易产生混淆 —— 这种情况其实可以避免。（直至正则表达式的重要性受到认可，其模式匹配机制才得以收录在标准程序库中。）
- 诸多工具程序具有缺陷，例如输入行（input lines）的长度，或是可打开的文件个数，等等。（现行的系统多半已经修正这些缺陷。）
- 有时程序并未经过彻底测试，这使得它们在执行的时候一不小心就会遭到破坏。这

注2： 我最初是在20世纪80年代从 Dan Forsyth 口中听到这个定义的。

可能会导致核心转储（core dumps，译注1），令用户不知所措。幸好，现行的UNIX系统极少会面临这样的问题。

- 系统的文档尽管大致上内容完备，但通常极为简单。使得用户在学习时很难找到所需要的信息（注3）。

本书之所以将重点放在文本（而非二进制）数据的处理与运用上，是由于UNIX早期的发展都源自于对文本处理的强烈需求，不过除此之外还有另外的重要理由（马上会讨论到）。事实上，贝尔实验室专利部门（Bell Labs Patent Department）在UNIX系统上所使用的第一套产品，就是进行文本处理和编排工作的。

最初的UNIX机器（Digital Equipment Corporation PDP-11s）不能运行大型程序。要完成复杂的工作，得先将它分割成更小的工作，再用程序来完成这些更小的工作。某些常见的工作（从数据行中取出某些字段、替换文本，等等）也常见于许多大型项目，最后就成了标准工具。人们认为这种自然而生的结果是件好事：由于缺乏大型的解决空间，因而产生了更小、更简单、更专用的程序。

许多人在UNIX的使用上采用半独立的工作方式，重复套用彼此间的程序。由于版本之间的差异，而且不需要标准化，导致许多日常工具程序的发展日渐分歧。举例来说，`grep`在某系统里使用`-i`来表示“查找时忽略大小写”，但在另一个系统中，却使用`-y`来代表同样的事！无独有偶，这种怪事也发生在许多工具程序上。还有，一些常用的小程序可能会取相同的名字，针对某个UNIX版本所编写的Shell程序，不经修改可能无法在另一个版本上执行。

最后，对常用标准工具组与选项的需求终于明朗化，POSIX标准即为最后的结果。现行标准IEEE Std. 1003.1-2004包含了C的库层级的主题，还有Shell语言与系统工具及其选项。

好消息是，在这些标准上所做的努力有了回报。现在的商用UNIX系统，以及可免费使

译注1：在UNIX系统中，常将“主内存”（main memory）称为核心（core），因为在使用半导体作为内存材料之前，便是使用核心（core）。而核心映像（core image）就是“进程”（process）执行当时的内存内容。当进程发生错误或收到“信号”（signal）而终止执行时，系统会将核心映像写入一个文件，以作为调试之用，这就是所谓的核心转储（core dump）。

注3：系统文档分成两个部分：参考手册与使用手册。后者是系统各功能的教学手册。虽然把整份文件读完就可能学会UNIX —— 事实上有许多人（包括作者）真的是这么做，不过现今的系统，已不再附上打印好的文件。

用的同类型产品，例如GNU/Linux与BSD衍生系统，都兼容POSIX。这样一来，学习UNIX变得更容易，编写可移植的Shell脚本也成为可能（详见第14章）。

值得注意的是：POSIX并非UNIX标准化的唯一成果，POSIX之外仍有其他标准。例如，欧洲计算机制造商协会自行发起了一套名为X/Open的标准。其中最受欢迎的是1988年首度出现的XPG4（X/Open Portability Guide，Fourth Edition）。另外还有XPG5，其更广为人知的名称为UNIX 98标准，或Single UNIX Specification。XPG5很大程度上把POSIX纳入为一个子集，同样深具影响力（注4）。

XPG标准在措辞上可能不够严谨，但其内容却较为广泛，其目标是将现存于UNIX系统上实际用到的各种功能正式生成文档。（POSIX的目的在于建立一套正式的标准，让从头开始的实践者有指导方针可以套用 —— 即便是在非UNIX的平台上。因此，许多UNIX系统上常见的功能，一开始就排除在POSIX标准之外）。2001 POSIX标准由于纳入了X/Open System Interface Extension（XSI）而有了双重身份，也叫做XPG6，这是它首度正式扩张POSIX版图。此文档的特色在于：让系统不只兼容POSIX，也兼容于XSI。所以，当你在编写工具或应用程序时，必须参考的正式文件只有一份（就叫做Single UNIX Standard）。

本书自始至终都把重点放在根据POSIX标准所定义的Shell语言与UNIX工具程序。重点部分也会加入XSI定义的说明，因为你很可能会用得到。

1.2 软件工具的原则

随着时间的流逝，人们开发出了一套设计与编写软件工具的原则。在本书用来解决问题的程序中，你将会看到这些原则的应用示例。好的软件工具应该具备下列特点：

一次做好一件事

在很多方面，这都是最重要的原则。若程序只做一件事，那么无论是设计、编写、调试、维护，以及生成文件都会容易得多。举例来说，对于用来查找文件中是否有符合样式的`grep`程序，不应该指望用它来执行算术运算。

这个原则的结果，自然就是会不断产生出更小、更专用于特定功能的程序，就像专业木匠的工具箱里，永远会有一堆专为特定用途所设计的工具。

处理文本行，不要处理二进制数据

文本行是UNIX的通用格式。当你在编写自己的工具程序时便会发现，内含文本行的数据文件很好处理，你可以用任何唾手可得的文本编辑器来编辑它，也可以让这

注4： X/Open的出版物列表可参见*http://www.opengroup.org/publications/catalog/*。

些数据在网络与各种机器架构之间传输。使用文本文件更有助于任何自定义工具与现存的UNIX程序之间的结合。

使用正则表达式

正则表达式（regular expression）是很强的文本处理机制。了解它的运作模式并加以使用，可适度简化编写命令脚本（script）的工作。

此外，虽然正则表达式多年来在工具与UNIX版本上不断在变化，但POSIX标准仅提供两种正则表达式。你可以利用标准的库程序进行模式匹配的工作。这样就可以编写出专用的工具程序，用于与`grep`一致的正则表达式（POSIX称之为基本型正则表达式，Basic Regular Expressions，BRE），或是用于与`egrep`一致的正则表达式（POSIX称之为扩展型正则表达式，Extended Regular Expressions，ERE）。

默认使用标准输入/输出

在未明确指定文件名的情况下，程序默认会从它的标准输入读取数据，将数据写到它的标准输出，至于错误信息则会传送到标准错误输出（这部分将于第2章讨论）。以这样的方式来编写程序，可以轻松地让它们成为数据过滤器（filter），例如，组成部分的规模越大，越需要复杂的管道（pipeline）或脚本来处理。

避免喋喋不休

软件工具的执行过程不该像在"聊天"（chatty）。不要将"开始处理"（`starting processing`）、"即将完成"（`almost done`）或是"处理完成"（`finished processing`）这类信息放进程序的标准输出（至少这不该是默认状态）。

当你有意将一些工具串成一条管道时，例如：

```
tool_1 < datafile | tool_2 | tool_3 | tool_4 > resultfile
```

若每个工具都会产生"正处理中"（yes I'm working）这样的信息并送往管道，那么别指望执行结果会像预期的一样。此外，若每个工具都将自己的信息传送至标准错误输出，那么整个屏幕画面就会布满一堆无用的过程信息。在工具程序的世界里，没有消息就是好消息。

这个原则其实还有另外一个含义。一般来说，UNIX工具程序一向遵循"你叫它做什么，你就会得到什么"的设计哲学。它们不会问"你确定吗？"（are you sure?）这种问题，当用户键入`rm somefile`，UNIX的设计人员会认为用户知道自己在做什么，然后毫无疑问地`rm`删除掉要删除的文件（注5）。

注5：　如果你真觉得这样不好，`rm`的`-i`选项可强制`rm`给你提示以做确认，这么一来，当你要求删除可疑文件时，`rm`便会提示确认它。一直以来，应该"永远不要提示"还是应该"永远得到提示"是个争议的话题，值得用户深思。

输出格式必须与可接受的输入格式一致

专业的工具程序认为遵循某种格式的输入数据，例如标题行之后接着数据行，或在行上使用某种字段分隔符等，所产生的输出也应遵循与输入一致的规则。这么做的好处是，容易将一个程序的执行结果交给另一个程序处理。

举例来说，netpbm程序集（注 5）是用来处理以 Portable BitMap（PBM）格式保存的图像文件（注 6）。这些文件内含bitmapped图像，并使用定义明确的格式加以绘制。每个读取PBM文件的工具程序，都会先以某种格式来处理文件内的图像，然后再以 PBM的格式写回文件。这么一来，便可以组合简单的管道来执行复杂的图像处理，例如先缩放影像后，再旋转方向，最后再把颜色调淡。

让工具去做困难的部分

虽然 UNIX 程序并非完全符合你的需求，但是现有的工具或许已经可以为你完成90%的工作。接下来，若有需要，你可以编写一个功能特定的小型程序来完成剩下的工作。与每次都从头开始来解决各个问题相比，这已经让你省去许多工作了。

构建特定工具前，先想想

如前所述，若现存系统里就是没有需要的程序，可以花点时间构建满足所需的工具。然而，动手编写一个能够解决问题的程序前，请先停下来想几分钟。你所要做的事，是否有其他人也需要做？这个特殊的工作是否有可能是某个一般问题的一个特例？如果是的话，请针对一般问题来编写程序。当然，这么做的时候，无论是在程序的设计或编写上，都应该遵循前面所提到的几项原则。

1.3 小结

UNIX 原为贝尔实验室的计算机科学家所开发的产品。由于没有盈利上的压力，再加上PDP-11小型计算机的能力有限，因而程序都以小型、优雅为圭臬。也因为没有盈利上的压力，系统之间并非完全一致，学习上也不太容易。

随着UNIX 持续地流行，各种版本陆续开发出来（尤其是衍生自 System V 和 BSD 的版本），Shell脚本层次的可移植性也日益困难。幸好，POSIX 标准成熟后，几乎所有商用UNIX 系统与免费的 UNIX 版本都兼容 POSIX。

注 6：这套程序并非 UNIX 工具集的标准配备，不过 GNU/Linux 与 BSD 系统上通常都会安装。其网站位于 *http://netpbm.sourceforge.net/*。可按照 Sourceforge 项目网页的指示，下载源代码。

注 7：有三种格式。若你的系统里有安装 netpbm，可参阅 *pnm*(5)手册页。

之所以会在这里指出软件工具的设计原则，主要是为了提供开发与使用UNIX工具集的指导方针。让软件工具的设计原则成为思考习惯，将有助于编写简洁的Shell程序和正确使用 UNIX 工具。

第2章 入门

当需要计算机帮你做些什么时，最好用对工具。你不会用文字编辑器来做支票簿的核对，也不会用计算器来写策划方案。同理，当你需要程序语言协助完成工作时，不同的程序语言用于不同的需求。

Shell 脚本最常用于系统管理工作，或是用于结合现有的程序以完成小型的、特定的工作。一旦你找出完成工作的方法，可以把用到的命令串在一起，放进一个独立的程序或脚本（script）里，此后只要直接执行该程序便能完成工作。此外，如果你写的程序很有用，其他人可以利用该程序当作一个黑盒（black box）来使用，它是一个可以完成工作的程序，但我们不必知道它是如何完成的。

本章中，我们会先对脚本编程（scripting）语言和编译型（compiled）语言做个简单的比较，再从如何编写简单的 Shell 脚本开始介绍起。

2.1 脚本编程语言与编译型语言的差异

许多中型、大型的程序都是用编译型语言写成，例如 Fortran、Ada、Pascal、C、C++ 或 Java。这类程序只要从源代码（source code）转换成目标代码（object code），便能直接通过计算机来执行（注 1）。

编译型语言的好处是高效，缺点则是：它们多半运作于底层，所处理的是字节、整数、浮点数或是其他机器层级的对象。例如，在 C++ 里，就很难进行“将一个目录里所有的文件复制到另一个目录中”之类的简单操作。

注 1：这种说法在 Java 上并不完全正确，不过已相当接近我们所说的情况了。

脚本编程语言通常是解释型(interpreted)的。这类程序的执行，是由解释器(interpreter)读入程序代码，并将其转换成内部的形式，再执行（注 2)。请注意，解释器本身是一般的编译型程序。

2.2 为什么要使用 Shell 脚本

使用脚本编程语言的好处是，它们多半运行在比编译型语言还高的层级，能够轻易处理文件与目录之类的对象。缺点是：它们的效率通常不如编译型语言。不过权衡之下，通常使用脚本编程还是值得的：花一个小时写成的简单脚本，同样的功能用C或C++来编写实现，可能需要两天，而且一般来说，脚本执行的速度已经够快了，快到足以让人忽略它性能上的问题。脚本编程语言的例子有 awk、Perl、Python、Ruby 与 Shell。

因为 Shell 似乎是各 UNIX 系统之间通用的功能，并且经过了 POSIX 的标准化。因此，Shell 脚本只要“用心写”一次，即可应用到很多系统上。因此，之所以要使用 Shell 脚本是基于：

简单性

Shell 是一个高级语言；通过它，你可以简洁地表达复杂的操作。

可移植性

使用 POSIX 所定义的功能，可以做到脚本无须修改就可在不同的系统上执行。

开发容易

可以在短时间内完成一个功能强大又好用的脚本。

2.3 一个简单的脚本

让我们从简单的脚本开始。假设你想知道，现在系统上有多少人登录。who 命令可以告诉你现在系统有谁登录：

```
$ who
george     pts/2      Dec 31 16:39    (valley-forge.example.com)
betsy      pts/3      Dec 27 11:07    (flags-r-us.example.com)
benjamin   dtlocal    Dec 27 17:55    (kites.example.com)
jhancock   pts/5      Dec 27 17:55    (:32)
camus      pts/6      Dec 31 16:22
tolstoy    pts/14     Jan  2 06:42
```

注 2： 尽管 *http://foldoc.doc.ic.ac.uk/foldoc/foldoc.cgi?Ousterhout's+dichotomy* 试图为编译型与脚本编程语言的差异下定义，但是人们对此一直很难达成共识。

在大型的、多用户的系统上，所列出来的列表可能很长，在你能够计算用户个数之前，列表早已滚动出屏幕画面，因此每次做这件事的时候，都会让你觉得很麻烦。这正是进行自动化的好时机。计算用户总数的方法尚未提到。对此，我们可以利用 wc（字数计算）程序，它可以算出行数（line）、字数（word）与字符数（character）。在此例中，我们用的是 wc -l，也就是只算出行数：

```
$ who | wc -l                    计算用户个数
      6
```

|（管道）符号可以在两程序之间建立管道（pipeline）：who 的输出，成了 wc 的输入，wc 所列出的结果就是已登录用户的个数。

下一步则是将此管道转变成一个独立的命令。方法是把这条命令输入一个一般的文件中，然后使用 chmod 为该文件设置执行的权限，如下所示：

```
$ cat > nusers                   建立文件，使用 cat 复制终端的输入
who | wc -l                      程序的内容
^D                               Ctrl-D 表示 end-of-file
$ chmod +x nusers                让文件拥有执行的权限
$ ./nusers                       执行测试
      6                          输出我们要的结果
```

这展现了小型 Shell 脚本的典型开发周期：首先，直接在命令行（command line）上测试。然后，一旦找到能够完成工作的适当语法，再将它们放进一个独立的脚本里，并为该脚本设置执行的权限。之后，就能直接使用该脚本。

2.4 自给自足的脚本：位于第一行的 #!

当 Shell 执行一个程序时，会要求 UNIX 内核启动一个新的进程（process），以便在该进程里执行所指定的程序。内核知道如何为编译型程序做这件事。我们的 nusers Shell 脚本并非编译型程序；当 Shell 要求内核执行它时，内核将无法做这件事，并回应“not executable format file”（不是可执行的格式文件）错误信息。Shell 收到此错误信息时，就会说“啊哈，这不是编译型程序，那么一定是 Shell 脚本”，接着会启动一个新的 /bin/sh（标准 Shell）副本来执行该程序。

当系统只有一个 Shell 时，“退回到 /bin/sh”的机制非常方便。但现行的 UNIX 系统都会拥有好几个 Shell，因此需要通过一种方式，告知 UNIX 内核应该以哪个 Shell 来执行所指定的 Shell 脚本。事实上，这么做有助于执行机制的通用化，让用户得以直接引用任何的程序语言解释器，而非只是一个命令 Shell。方法是，通过脚本文件中特殊的第一行来设置：在第一行的开头处使用 #! 这两个字符。

当一个文件中开头的两个字符是 #! 时，内核会扫描该行其余的部分，看是否存在可用来执行程序的解释器的完整路径。（中间如果出现任何空白符号都会略过。）此外，内核还会扫描是否有一个选项要传递给解释器。内核会以被指定的选项来引用解释器，再搭配命令行的其他部分。举例来说，假设有一个 csh 脚本（注 3），名为 /usr/ucb/whizprog，它的第一行如下所示：

```
#! /bin/csh -f
```

再者，如果 Shell 的查找路径（后面会介绍）里有 /usr/ucb，当用户键入 whizprog -q /dev/tty01 这条命令，内核解释 #! 这行后，便会以如下的方式来引用 csh：

```
/bin/csh -f /usr/ucb/whizprog -q /dev/tty01
```

这样的机制让我们得以轻松地引用任何的解释器。例如我们可以这样引用独立的 awk 程序：

```
#! /bin/awk -f
此处是 awk 程序
```

Shell 脚本通常一开始都是 #! /bin/sh。如果你的 /bin/sh 并不符合 POSIX 标准，请将这个路径改为符合 POSIX 标准的 Shell。下面是几个初级的陷阱（gotchas），请特别留意：

- 当今的系统，对 #! 这一行的长度限制从 63 到 1024 个字符（character）都有。请尽量不要超过 64 个字符。（表 2-1 列出了各系统的长度限制。）
- 在某些系统上，命令行部分（也就是要传递给解释器执行的命令）包含了命令的完整路径名称。不过有些系统却不是这样；命令行的部分会原封不动地传给程序。因此，脚本是否具可移植性取决于是否有完整的路径名称。
- 别在选项（option）之后放置任何空白，因为空白也会跟着选项一起传递给被引用的程序。
- 你需要知道解释器的完整路径名称。这可以用来规避可移植性问题，因为不同的厂商可能将同样的东西放在不同的地方（例如 /bin/awk 和 /usr/bin/awk）。
- 一些较旧的系统上，内核不具备解释 #! 的能力，有些 Shell 会自行处理，这些 Shell 对于 #! 与紧随其后的解释器名称之间是否可以有空白，可能有不同的解释。

注 3： /bin/csh 是 C Shell 的命令解释器，由加州大学伯克利分校所开发。本书不讨论 C Shell 程序设计的原因很多，其中最重要一的点是：就脚本的编写来说，大多数人认为它不是个好用的 Shell，另一个原因则是它并未被 POSIX 标准化。

表 2-1 列出了各 UNIX 系统对于 #!行的长度限制（这些都是通过经验法则得知的）。结果出乎意料：有一半以上的数字都不是二的次方。

表 2-1：各系统对 #!行的长度限制

平台	操作系统版本	最大长度
Apple Power Mac	Mac Darwin 7.2 (Mac OS 10.3.2)	512
Compaq/DEC Alpha	OSF/1 4.0	1024
Compaq/DEC/HP Alpha	OSF/1 5.1	1000
GNU/Linux 注	Red Hat 6, 7, 8, 9; Fedora 1	127
HP PA-RISC and Itanium-2	HP-UX 10, 11	127
IBM RS/6000	AIX 4.2	255
Intel x86	FreeBSD 4.4	64
Intel x86	FreeBSD 4.9, 5.0, 5.1	128
Intel x86	NetBSD 1.6	63
Intel x86	OpenBSD 3.2	63
SGI MIPS	IRIX 6.5	255
Sun SPARC, x86	Solaris 7, 8, 9, 10	1023

注：所有架构。

POSIX 标准对 #!的行为模式保留未定义（unspecified）状态。此状态是“只要一直保持 POSIX 兼容性，这是一个扩展功能”的标准说法。

本书接下来的所有脚本开头都会有 #!行。下面是修订过的 nusers 程序：

```
$ cat nusers                    显示文件内容
#! /bin/sh -                    神奇的 #!行

who | wc -l                     所要执行的命令
```

选项 —— 表示没有 Shell 选项 ；这是基于安全上的考虑，可避免某种程度的欺骗式攻击（spoofing attack）。

2.5 Shell 的基本元素

本节要介绍的是，适用于所有 Shell 脚本的基本元素。通过以交互的方式使用 Shell，你会慢慢熟悉的。

2.5.1 命令与参数

Shell最基本的工作就是执行命令。以互动的方式来使用Shell很容易了解这一点：每键入一道命令，Shell就会执行。像这样：

```
$ cd work ; ls -l whizprog.c
-rw-r--r--    1 tolstoy   devel       30252 Jul  9 22:52 whizprog.c
$ make
...
```

以上的例子展现了UNIX命令行的原理。首先，格式很简单，以空白（Space键或Tab键）隔开命令行中各个组成部分。

其次，命令名称是命令行的第一个项目。通常后面会跟着选项（option），任何额外的参数（argument）都会放在选项之后。如下的语法是不可能出现的：

```
COMMAND=CD,ARG=WORK
COMMAND=LISTFILES,MODE=LONG,ARG=WHIZPROG.C
```

这类语法多半出现在正在设计UNIX时的传统大型系统上。UNIX Shell的自由格式语法在当时是一大革新，大大增强了Shell脚本的可读性。

第三，选项的开头是一个破折号（或减号），后面接着一个字母。选项是可有可无的，有可能需要加上参数（例如cc -o whizprog whizprog.c）。不需要参数的选项可以合并：例如，ls -lt whizprog.c比ls -l -t whizprog.c更方便（后者当然也可以，只是得多些录入）。

长选项的使用越来越普遍，特别是标准工具的GNU版本，以及在X Window System（X11）下使用的程序。例如：

```
$ cd whizprog-1.1
$ patch --verbose --backup -p1 < /tmp/whizprog-1.1-1.2-patch
```

长选项的开头是一个破折号还是两个（如上所示），视程序而定。（< /tmp/whizprog-1.1-1.2-patch是一个I/O重定向。它会使得patch从/tmp/whizprog-1.1-1.2-patch文件而不是从键盘读取输入。I/O重定向也是重要的基本概念之一，本章稍后会谈到）。

以两个破折号（--）来表示选项结尾的用法，源自System V，不过已被纳入POSIX标准。自此之后命令行上看起来像选项的任何项目，都将一视同仁地当成参数处理（例如，视为文件名）。

最后要说的是，分号（;）可用来分隔同一行里的多条命令。Shell会依次执行这些命令。

如果你使用的是 & 符号而不是分号，则 Shell 将在后台执行其前面的命令，这意味着，Shell 不用等到该命令完成，就可以继续执行下一个命令。

Shell 识别三种基本命令：内建命令、Shell 函数以及外部命令：

- 内建命令就是由 Shell 本身所执行的命令。有些命令是由于其必要性才内建的，例如 cd 用来改变目录，read 会将来自用户（或文件）的输入数据传给 Shell 变量。另一种内建命令的存在则是为了效率，其中最典型的就是 test 命令（稍后在 6.2.4 节会谈到），编写脚本时会经常用到它。另外还有 I/O 命令，例如 echo 与 printf。
- Shell 函数是功能健全的一系列程序代码，以 Shell 语言写成，它们可以像命令那样引用。稍后会在 6.5 节讨论这个部分。此处，我们只需要知道，它们可以引用，就像一般的命令那样。
- 外部命令就是由 Shell 的副本（新的进程）所执行的命令，基本的过程如下：
    a. 建立一个新的进程。此进程即为 Shell 的一个副本。
    b. 在新的进程里，在 PATH 变量内所列出的目录中，寻找特定的命令。/bin:/usr/bin:/usr/X11R6/bin:/usr/local/bin 为 PATH 变量典型的默认值。当命令名称含有斜杠（/）符号时，将略过路径查找步骤。
    c. 在新的进程里，以所找到的新程序取代执行中的 Shell 程序并执行。
    d. 程序完成后，最初的 Shell 会接着从终端读取的下一条命令，或执行脚本里的下一条命令。如图 2-1 所示。

以上只是基本程序。当然，Shell 可以做的事很多，例如变量与通配字符的展开、命令与算术的替换等。接下来，本书会一一探讨这些话题。

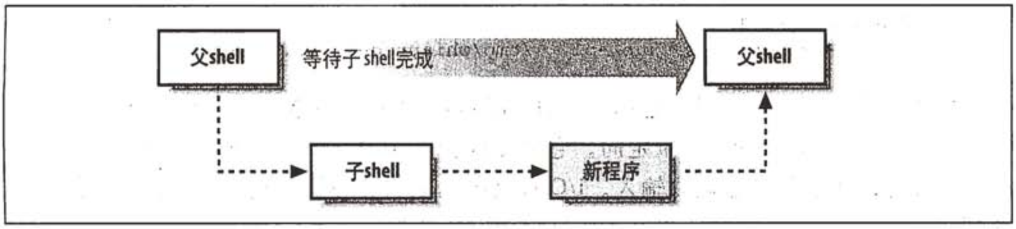


图 2-1：程序执行

2.5.2 变量

变量（variable）就是为某个信息片段所起的名字，例如 first_name 或 driver_lic_no。所有程序语言都会有变量，Shell 也不例外。每个变量都有一个值（value），这是由你分

配给变量的内容或信息。在Shell的世界里，变量值可以是（而且通常是）空值，也就是不含任何字符。这是合理的，也是常见的、好用的特性。空值就是null，本书接下来的部分将会经常用到这个术语。

Shell变量名称的开头是一个字母或下划线符号，后面可以接着任意长度的字母、数字或下划线符号。变量名称的字符长度并无限制。Shell变量可用来保存字符串值，所能保存的字符数同样没有限制。Bourne Shell是少数几个早期的UNIX程序里，遵循不限制（no arbitrary limit）设计原则的程序之一。例如：

```
$ myvar=this_is_a_long_string_that_does_not_mean_much     分配变量值
$ echo $myvar                                             打印变量值
this_is_a_long_string_that_does_not_mean_much
```

变量赋值的方式为：先写变量名称，紧接着=字符，最后是新值，中间完全没有任何空格。当你想取出Shell变量的值时，需于变量名称前面加上$字符。当所赋予的值内含空格时，请加上引号：

```
first=isaac  middle=bashevis  last=singer    单行可进行多次赋值
fullname="isaac bashevis singer"             值中包含空格时使用引号
oldname=$fullname                            此处不需要引号
```

如上例所示，当变量作为第二个变量的新值时，不需要使用双引号（参见7.7节），但是使用双引号也没关系。不过，当你将几个变量连接起来时，就需要使用引号了：

```
fullname="$first $middle $last"              这里需要双引号
```

2.5.3 简单的echo输出

这里要看的是echo命令如何显示myvar变量的值，这是很可能会在命令行里使用到的情况。echo的任务就是产生输出，可用来提示用户，或是用来产生数据供进一步处理。

原始的echo命令只会将参数打印到标准输出，参数之间以一个空格隔开，并以换行符号（newline）结尾。

```
$ echo Now is the time for all good men
Now is the time for all good men
$ echo to come to the aid of their country.
to come to the aid of their country.
```

不过，随着时间的流逝，有各种版本的echo开发出来。BSD版本的echo看到第一个参数为-n时，会省略结尾的换行符号。例如（下划线符号表示终端画面的光标）：

```
$ echo -n "Enter your name: "                显示提示
Enter your name: _                           键入数据
```

echo

语法

echo [*string* ...]

用途

产生Shell脚本的输出。

主要选项

无。

行为模式

echo将各个参数打印到标准输出，参数之间以一个空格隔开，并以换行符号结束。它会解释每个字符串里的转义序列（escape sequences）。转义序列可用来表示特殊字符，以及控制其行为模式。

警告

UNIX各版本间互不相同的行为模式使得echo的可移植性变得很困难，不过它仍是最简单的一种输出方式。

许多版本都支持-n选项。如果有支持，echo的输出会省略最后的换行符号。这适合用来打印提示字符串。不过，目前echo符合POSIX标准的版本并未包含此选项。

System V版本的echo会解释参数里特殊的转义序列（稍后会说明）。例如，\c用来指示echo不要打印最后的换行符号：

```
$ echo "Enter your name: \c"          显示提示
Enter your name: _                    键入数据
```

转义序列可用来表示程序中难以键入或难以看见的字符。echo遇到转义序列时，会打印相应的字符。有效的转义序列如表2-2所示。

表2-2：echo的转义序列

序列	说明
\a	警示字符，通常是ASCII的BEL字符
\b	退格（Backspace）
\c	输出中忽略最后的换行字符（Newline）。这个参数之后的任何字符，包括接下来的参数，都会被忽略掉（不打印）
\f	清除屏幕（Formfeed）
\n	换行（Newline）

表 2-2：echo 的转义序列（续）

序列	说明
`\r`	回车（Carriage return）
`\t`	水平制表符（Horizontal tab）
`\v`	垂直制表符（Vertical tab）
`\\`	反斜杠字符
`\0ddd`	将字符表示成 1 到 3 位的八进制数值

实际编写 Shell 脚本的时候，`\a` 序列通常用来引起用户的注意；`\0ddd` 序列最有用的地方，就是通过送出终端转义序列进行（非常）原始的光标操作，但是不建议这么做。

由于很多系统默认以BSD的行为模式来执行`echo`，所以本书只会使用它的最简单形式。比较复杂的输出，我们会使用`printf`。

2.5.4 华丽的 printf 输出

由于`echo`有版本上的差异，所以导致 UNIX 版本间可移植性的头疼问题。在 POSIX 标准化的首次讨论中，与会成员无法在如何标准化`echo`上达到共识，便提出折衷方案。虽然`echo`是 POSIX 标准的一部分，但该标准却未说明当第一个参数是`-n`或有任何参数包含转义序列的行为模式。取而代之的是，将其行为模式保留为实现时定义（implementation-defined），也就是说，各厂商必须提供说明文件，描述其`echo`版本的做法（注 4）。事实上，只要是使用最简单的形式，其`echo`的可移植性不会有问题。相对来看，Ninth Edition Research UNIX 系统上所采用的`printf`命令，比`echo`更灵活，却也更复杂。

`printf`命令模仿 C 程序库（library）里的`printf()`库程序（library routine）。它几乎复制了该函数所有的功能（见*printf(3)*的在线说明文档），如果你曾使用C、C++、awk、Perl、Python 或 Tcl 写过程序，对它的基本概念应该不陌生。当然，它在 Shell 层级的版本上，会有些差异。

如同`echo`命令，`printf`命令可以输出简单的字符串：

```
printf "Hello, world\n"
```

你应该可以马上发现，最大的不同在于：`printf`不像`echo`那样会自动提供一个换行符号。你必须显式地将换行符号指定成`\n`。`printf`命令的完整语法分为两部分：

```
printf format-string [arguments ...]
```

注 4： 值得玩味的是，现行版本的标准中，说明`echo`在本质上等同于 System V 版本，后者会处理其参数中的转义序列，但不处理`-n`。

第一部分是一个字符串，用来描述输出的排列方式，最好为此字符串加上引号。此字符串包含了按字面显示的字符（characters to be printed literally）以及格式声明（format specifications），后者是特殊的占位符（placeholders），用来描述如何显示相应的参数（argument）。

第二部分是与格式声明相对应的参数列表（argument list），例如一系列的字符串或变量值。（如果参数的个数比格式声明还多，则 printf 会循环且依次地地使用格式字符串里的格式声明，直到处理完参数）。格式声明分成两部分：百分比符号（%）和指示符（specifier）。最常用的格式指示符（format specifier）有两个，%s 用于字符串，而 %d 用于十进制整数。

格式字符串中，一般字符会按字面显示。转义序列则像 echo 那样，解释后再输出成相应的字符。格式声明以 % 符号开头，并以定义的字母集中的一个来结束，用来控制相应参数的输出。例如，%s 用于字符串的输出：

```
$ printf "The first program always prints '%s, %s!'\n" Hello world
The first program always prints 'Hello, world!'
```

printf 的所有详细说明见 7.4 节。

2.5.5 基本的 I/O 重定向

标准输入/输出（standard I/O，注 5）可能是软件设计原则里最重要的概念了。这个概念就是：程序应该有数据的来源端、数据的目的端（数据要去的地方）以及报告问题的地方，它们分别被称为标准输入（standard input）、标准输出（standard output）以及标准错误输出（standard error）。程序不必知道也不用关心它的输入与输出背后是什么设备：是磁盘上的文件、终端、磁带机、网络连接或是另一个执行中的程序！当程序启动时，可以预期的是，标准输出入都已打开，且已准备好供其使用。

许多 UNIX 程序都遵循这一设计原则。默认的情况下，它们会读取标准输入、写入标准输出，并将错误信息传递到标准错误输出。这类程序常叫做过滤器（filter），你马上就会知道这么叫的原因。默认的标准输入、标准输出以及标准错误输出都是终端，这点可通过 cat 程序得知：

```
$ cat                              未指定任何参数，读取标准输入，写入标准输出
now is the time                    由用户键入
now is the time                    由 cat 返回
for all good men
for all good men
to come to the aid of their country
```

注 5：此处的 Standard I/O 请不要与 C 程序库的 standard I/O 程序库混淆了，后者的接口定义于 <stdio.h>，不过此程序库的工作一样是提供类似的概念给 C 程序使用。

```
to come to the aid of their country
^D                                   Ctrl-D，文件结尾
```

你可能想要知道，是谁替执行中的程序初始化标准输入、输出及错误输出的呢？毕竟，总应该有人来替执行中的程序打开这些文件，甚至是让用户在登录后能够看到交互的Shell界面。

答案就是在你登录时，UNIX便将默认的标准输入、输出及错误输出安排成你的终端。I/O重定向就是你通过与终端交互，或是在Shell脚本里设置，重新安排从哪里输入或输出到哪里。

2.5.5.1 重定向与管道

Shell提供了数种语法标记，可用来改变默认I/O的来源端与目的端。此处会先介绍基本用法，稍后再提供完整的说明。让我们由浅入深地依次介绍如下：

以 < 改变标准输入

program < *file*可将*program*的标准输入修改为*file*：

```
tr -d '\r' < dos-file.txt ...
```

以 > 改变标准输出

program > *file*可将*program*的标准输出修改为*file*：

```
tr -d '\r' < dos-file.txt > UNIX-file.txt
```

这条命令会先以tr将dos-file.txt里的ASCII carriage-return（回车）删除，再将转换完成的数据输出到UNIX-file.txt。dos-file.txt里的原始数据不会有变化。（tr命令在第5章有完整的说明。）

>重定向符（redirector）在目的文件不存在时，会新建一个。然而，如果目的文件已存在，它就会被覆盖掉；原本的数据都会丢失。

以 >> 附加到文件

program >> *file*可将*program*的标准输出附加到*file*的结尾处。

如同>，如果目的文件不存在，>>重定向符便会新建一个。然而，如果目的文件存在，它不会直接覆盖掉文件，而是将程序所产生的数据附加到文件结尾处：

```
for f in dos-file*.txt
do
    tr -d '\r' < $f >> big-UNIX-file.txt
done
```

（for循环的介绍详见6.4节。）

以 | 建立管道

program1 | *program2*可将*program1*的标准输出修改为*program2*的标准输入。

虽然 < 与 > 可将输入与输出连接到文件，不过管道（pipeline）可以把两个以上执行中的程序衔接在一起。第一个程序的标准输出可以变成第二个程序的标准输入。这么做的好处是，管道可以使得执行速度比使用临时文件的程序快上十倍。本书中有相当多篇幅都是在讨论如何将各类工具串在一起，置入越来越复杂且功能越来越强大的管道中。例如：

```
tr -d '\r' < dos-file.txt | sort > UNIX-file.txt
```

这条管道会先删除输入文件内的回车字符，在完成数据的排序之后，将结果输出到目的文件。

tr

语法

tr [*options*] *source-char-list replace-char-list*

用途

转换字符。例如，将大写字符转换成小写。选项可让你指定所要删除的字符，以及将一串重复出现的字符浓缩成一个。

常用选项

-c

取 source-char-list 的反义。tr 要转换的字符，变成未列在 source-char-list 中的字符。此选项通常与 -d 或 -s 配合使用。

-C

与 -c 相似，但所处理的是字符（可能是包含多个字节的宽字符），而非二进制的字节值。参考“警告”的说明。

-d

自标准输入删除 *source-char-list* 里所列的字符，而不是转换它们。

-s

浓缩重复的字符。如果标准输入中连续重复出现 *source-char-list* 里所列的字符，则将其浓缩成一个。

行为模式

如同过滤器：自标准输入读取字符，再将结果写到标准输出。任何输入字符只要出现在 *source-char-list* 中，就会置换成 *replace-char-list* 里相应的字符。POSIX风格的字符与等效的字符集也适用，而且tr还支持replace-char-list 中重复字符的标记法。相关细节请参考 *tr*(1)的在线说明文档。

警告

根据POSIX标准的定义，-c处理的是二进制字节值，而 -C处理的是现行locale所定义的字符。直到 2005 年初，仍有许多系统不支持 -C 选项。

使用UNIX工具程序时，不妨将数据想象成水管里的水。未经处理的水，将流向净水厂，经过各类滤器的处理，最后产生适合人类饮用的水。

同样，编写脚本时，你通常已有某种输入格式定义下的原始数据，而需要处理这些数据后产生结果。(处理一词表示很多意思，例如排序、加和与平均、格式化以便于打印，等等。）从最原始的数据开始，然后构造一条管道，一步一步地，管道中的每个阶段都会让数据更接近要的结果。

如果你是UNIX新手，可以把 < 与 > 想象成数据的漏斗（funnels）—— 数据会从大的一端进入，由小的一端出来。

注意： 构造管道时，应该试着让每个阶段的数据量变得更少。换句话说，如果你有两个要完成的步骤与先后次序无关，你可以把会让数据量变少的那一个步骤放在管道的前面。这么做可以提升脚本的整体性能，因为UNIX只需要在两个程序间移动少的数据量，每个程序要做的事也比较少。

例如，使用 `sort` 排序之前，先以 `grep` 找出相关的行；这样可以让 `sort` 少做些事。

2.5.5.2 特殊文件：/dev/null 与 /dev/tty

UNIX系统提供了两个对Shell编程特别有用的特殊文件。第一个文件 `/dev/null`，就是大家所熟知的位桶（bit bucket）。传送到此文件的数据都会被系统丢掉。也就是说，当程序将数据写到此文件时，会认为它已成功完成写入数据的操作，但实际上什么事都没做。如果你需要的是命令的退出状态（见6.2节），而非它的输出，此功能会很有用。例如，测试一个文件是否包含某个模式（pattern）：

```
if grep pattern myfile > /dev/null
then
    ...      找到模式时
else
    ...      找不到模式时
fi
```

相对地，读取 `/dev/null` 则会立即返回文件结束符号（end-of-file）。读取 `/dev/null` 的操作很少会出现在Shell程序里，不过了解这个文件的行为模式还是非常重要的。

另一个特殊文件为 `/dev/tty`。当程序打开此文件时，UNIX会自动将它重定向到一个终端［一个实体的控制台（console）或串行端口（serial port），也可能是一个通过网络与窗口登录的伪终端（pseudoterminal）］再与程序结合。这在程序必须读取人工输入时（例如密码）特别有用。此外，用它来产生错误信息也很方便，只是比较少人这么做：

```
printf "Enter new password: "    提示输入
stty -echo                       关闭自动打印输入字符的功能
read pass < /dev/tty             读取密码
printf "Enter again: "           提示再输入一次
read pass2 < /dev/tty            再读取一次以确认
stty echo                        别忘了打开自动打印输入字符的功能
...
```

stty (set tty) 命令用来控制终端（或窗口，注 6）的各种设置。-echo 选项用来关闭自动打印每个输入字符的功能；stty echo 用来恢复该功能。

2.5.6 基本命令查找

之前，我们曾提及 Shell 会沿着查找路径 $PATH 来寻找命令。$PATH 是一个以冒号分隔的目录列表，你可以在列表所指定的目录下找到所要执行的命令。所找到的命令可能是编译后的可执行文件，也可能是 Shell 脚本；从用户的角度来看，两者并无不同。

默认路径（default path）因系统而异，不过至少包含 /bin 与 /usr/bin，或许还包含存放 X Windows 程序的 /usr/X11R6/bin，以及供本地系统管理人员安装程序的 /usr/local/bin。例如：

```
$ echo $PATH
/bin:/usr/bin:/usr/X11R6/bin:/usr/local/bin
```

名称为 bin 的目录用来保存可执行文件，bin 是 binary 的缩写。你也可以直接把 bin 解释成相应的英文字义 —— 存储东西的容器；这里所存储的是可执行的程序。

如果你要编写自己的脚本，最好准备自己的 bin 目录来存放它们，并且让 Shell 能够自动找到它们。这不难，只要建立自己的 bin 目录，并将它加入 $PATH 中的列表即可：

```
$ cd                          切换到 home 目录
$ mkdir bin                   建立个人 bin 目录
$ mv nusers bin               将我们的脚本置入该目录
$ PATH=$PATH:$HOME/bin        将个人的 bin 目录附加到 PATH
$ nusers                      试试看
      6                       Shell 有找到并执行它
```

要让修改永久生效，在 .profile 文件中把你的 bin 目录加入 $PATH，而每次登录时 Shell 都将读取 .profile 文件，例如：

```
PATH=$PATH:$HOME/bin
```

注 6： stty 可能是现有的 UNIX 命令中，最怪异且最复杂的一个。相关细节可参考 *stty*(1) 的 manpage 或是《UNIX in a Nutshell》这本书。

$PATH 里的空项目（empty component）表示当前目录（current directory）。空项目位于路径值中间时，可以用两个连续的冒号来表示。如果将冒号直接置于最前端或尾端，可以分别表示查找时最先查找或最后查找当前目录：

```
PATH=:/bin:/usr/bin:/usr/X11R6/bin:/usr/local/bin      先找当前目录
PATH=/bin:/usr/bin:/usr/X11R6/bin:/usr/local/bin:      最后找当前目录
PATH=/bin:/usr/bin:/usr/X11R6/bin::/usr/local/bin      当前目录居中
```

如果你希望将当前目录纳入查找路径（search path），更好的做法是在 $PATH 中使用点号（dot）；这可以让阅读程序的人更清楚程序在做什么。

测试过程中，我们发现同一个系统有两个版本并未正确支持$PATH结尾的空项目，因此空项目在可移植性上有点问题。

注意： 一般来说，你根本就不应该在查找路径中放进当前目录，因为这会有安全上的问题（进一步的信息请参考第15章）。之所以会提到空项目，只是为了让你了解路径查找的运作模式。

2.6 访问 Shell 脚本的参数

所谓的位置参数（positional parameters）指的也就是Shell脚本的命令行参数（command-line arguments）。在 Shell 函数里，它们同时也可以是函数的参数。各参数都由整数来命名。基于历史的原因，当它超过 9，就应该用大括号把数字框起来：

```
echo first arg is $1
echo tenth arg is ${10}
```

此外，通过特殊变量，我们还可以取得参数的总数，以及一次取得所有参数。相关细节参见 6.1.2.2 节。

假设你想知道某个用户正使用的终端是什么，你当然可以直接使用who命令，然后在输出中自己慢慢找。这么做很麻烦又容易出错 —— 特别是当系统的用户很多的时候。你想做的只不过是在who的输出中找到那位用户，这个时候你可以用grep命令来进行查找操作，它会列出与第一个参数（所指定的模式）匹配的每一行。假设你要找的是用户betsy：

```
$ who | grep betsy                              betsy 在哪？
betsy      pts/3        Dec 27 11:07     (flags-r-us.example.com)
```

知道如何寻找特定的用户后，我们可以将命令放进脚本里，这段脚本的第一个参数就是我们要找的用户名称：

```
$ cat > finduser                              建立新文件
#! /bin/sh

# finduser --- 察看第一个参数所指定的用户是否登录

who | grep $1
^D                                            以End-of-file结尾

$ chmod +x finduser                           设置执行权限

$ ./finduser betsy                            测试：寻找 betsy
betsy      pts/3        Dec 27 11:07          (flags-r-us.example.com)

$ ./finduser benjamin                         再找找好友Ben
benjamin   dtlocal      Dec 27 17:55          (kites.example.com)

$ mv finduser $HOME/bin                       将这个文件存进自己的bin目录
```

以# finduser...开头的这一行是一个注释（comment）。Shell会忽略由#开头的每一行。（相信你也已经发现：当Shell读取脚本时，前面所提及的#!行也同样扮演注释的角色。）为你的程序加上注释绝对不会错。这样可以帮助其他人或是自己在一年以后还能够了解你在做什么以及为什么要这么做。等到我们觉得程序能够运行无误时，就可以把它移到个人的bin目录。

这个程序还没有达到完美。要是我们没给任何参数，会发生什么事？

```
$ finduser
Usage: grep [OPTION]... PATTERN [FILE]...
Try 'grep --help' for more information.
```

我们将在6.2.4节看到，如何测试命令行参数数目，以及在参数数目不符时，如何采取适当的操作。

2.7 简单的执行跟踪

程序是人写的，难免会出错。想知道你的程序正在做什么，有个好方法，就是把执行跟踪（execution tracing）的功能打开。这会使得Shell显示每个被执行到的命令，并在前面加上“+”：一个加号后面跟着一个空格。（你可以通过给Shell变量PS4赋一个新值以改变打印方式。）

例如：

```
$ sh -x nusers                打开执行跟踪功能
+ who                         被跟踪的命令
+ wc -l
      7                       实际的输出
```

你可以在脚本里，用 set -x 命令将执行跟踪的功能打开，然后再用 set +x 命令关闭它。这个功能对复杂的脚本比较有用，不过这里只用简单的程序来做说明：

```
$ cat > trace1.sh                    建立脚本
#! /bin/sh

set -x                               打开跟踪功能
echo 1st echo                        做些事

set +x                               关闭跟踪功能
echo 2nd echo                        再做些事
^D                                   以 end-of-file 结尾

$ chmod +x trace1.sh                 设置执行权限

$ ./trace1.sh                        执行
+ echo 1st echo                      被跟踪的第一行
1st echo                             命令的输出
+ set +x                             被跟踪的下一行
2nd echo                             下一个命令的输出
```

执行时，set -x 不会被跟踪，因为跟踪功能是在这条命令执行后才打开的。同理，set +x 会被跟踪，因为跟踪功能是在这条命令执行后才关闭的。最后的 echo 命令不会被跟踪，因为此时跟踪功能已经关闭。

2.8 国际化与本地化

编写软件给全世界的人使用，是一项艰难的挑战。整个工作通常可以分成两个部分：国际化（internationalization，缩写为 i18n，因为这个单字在头尾之间包含了 18 个字母），以及本地化（localization，缩写为 l10n，理由同前）。

当国际化作为设计软件的过程时，软件无须再修改或重新编译程序代码，就可以给特定的用户群使用。至少这表示，你必须将“所要显示的任何信息”包含在特定的程序库调用里，执行期间由此“程序库调用”负责在消息目录（message catalog）中找到适当的译文。一般来说，消息的译文就放在软件附带的文本文件中，再通过 gencat 或 msgfmt 编译成紧凑的二进制文件，以利快速查询。编译后的信息文件会被安装到特定的系统目录树中，例如 GNU 的 /usr/share/locale 与 /usr/local/share/locale，或商用 UNIX 系统的 /usr/lib/nls 或 /usr/lib/locale。详情可见 setlocale(3)、catgets(3C) 与 gettext(3C) 等手册页面（manual pages）。

当本地化作为设计软件的过程时，目的是让特定的用户群得以使用软件。在本地化的过程可能需要翻译软件文件和软件所输出的所有文字，可能还必须修改程序输出中的货币、日期、数字、时间、单位换算等格式。文字所使用的字符集（character set）可能也得

变动（除非使用通用的Unicode字符集），并且使用不同的字体。对某些语言来说，书写方向（writing direction）也可能需要变动。

UNIX的世界中，ISO程序语言标准与POSIX对此类问题的处理都提供了有限度的支持，不过要做的事还很多，而且各种UNIX版本之间差异极大。对用户而言，用来控制让哪种语言或文化环境生效的功能就叫做*locale*，你可以通过如表2-3所示的一个或多个环境变量（environment variable）来设置它。

表2-3：各种Locale环境变量

名称	说明
`LANG`	未设置任何`LC_xxx`变量时所使用的默认值
`LC_ALL`	用来覆盖掉所有其他`LC_xxx`变量的值
`LC_COLLATE`	使用所指定地区的排序规则
`LC_CTYPE`	使用所指定地区的字符集（字母、数字、标点符号等）
`LC_MESSAGES`	使用所指定地区的响应与信息；仅POSIX适用
`LC_MONETARY`	使用所指定地区的货币格式
`LC_NUMERIC`	使用所指定地区的数字格式
`LC_TIME`	使用所指定地区的日期与时间格式

一般来说，你可以用`LC_ALL`来强制设置单一locale；而`LANG`则是用来设置locale的默认值。大多数时候，应避免为任何的`LC_xxx`变量赋值。举例来说，当你使用`sort`命令时，可能会出现要你正确设置`LC_COLLATE`的信息，因为这个设置可能会跟`LC_CTYPE`的设置相冲突，也可能在`LC_ALL`已设置的情况下完全被忽略。

ISO C与C++标准只定义了`C`这个标准的locale名称：用来选择传统的面向ASCII的行为模式。POSIX标准则另外定义了`POSIX`这个locale名称，其功能等同于`C`。

除`C`与`POSIX`外，locale名称并未标准化。不过，有很多厂商采用类似但不一致的名称。locale名称带有语言和地域的意义，有时甚至会加上一个内码集（codeset）与一个修饰符（modifier）。一般来说，它会被表示成ISO 639语言代码（language code，注7）的两个小写字母、一个下划线符号与ISO 3166-1国家代码（country code，注8）的两个大写字母，最后可能还会加上一个点号、字符集编码、@符号与修饰词（modifier word）。语文名称有时也会用上。你可以像下面这样列出系统认得哪些locale名称：

注7：　见 *http://www.ics.uci.edu/pub/ietf/http/related/iso639.txt*。

注8：　见 *http://userpage.chemie.fu-berlin.de/diverse/doc/ISO_3166.html*。

```
$ locale -a                                    列出所有 locale 名称
...
français
fr_BE
fr_BE@euro
fr_BE.iso88591
fr_BE.iso885915@euro
fr_BE.utf8
fr_BE.utf8@euro
fr_CA
fr_CA.iso88591
fr_CA.utf8
...
french
...
```

查询特定locale变量相关细节的方法如下：为执行环境指定locale（放在命令前面）并以 `-ck` 选项与一个 `LC_xxx` 变量来执行 `locale` 命令。下面的例子是在Sun Solaris系统下，以Danish（丹麦文）locale来查询日期时间格式所得到结果：

```
$ LC_ALL=da locale -ck LC_TIME          取得Danish的日期时间格式
 LC_TIME
d_t_fmt="%a %d %b %Y %T %Z"
d_fmt="%d-%m-%y"
t_fmt="%T"
t_fmt_ampm="%I:%M:%S %p"
am_pm="AM";"PM"
day="sØn dag";"mandag";"tirsdag";"onsdag";"torsdag";"fredag";"lØrdag"
abday="sØn ";"man";"tir";"ons";"tor";"fre";"lØr"
mon="januar";"februar";"marts";"april";"maj";"juni";"juli";"august"; \
    "september";"oktober";"november";"december"
abmon="jan";"feb";"mar";"apr";"maj";"jun";"jul";"aug";"sep";"okt"; \
      "nov";"dec"
era=""
era_d_fmt=""
era_d_t_fmt=""
era_t_fmt=""
alt_digits=""
```

能够使用的locale相当多。一份调查了约20种UNIX版本的报告发现，BSD与Mac OS X系统完全不支持locale（没有 `locale` 命令可用），甚至在某些系统上也只支持5种，不过新近发布的GNU/Linux版本则几乎可以支持500种。locale的支持在安装时或许可以由系统管理者自行决定，所以即便是相同的操作系统，安装在两个类似的机器上，对locale的支持可能有所不同。我们发现，在某些系统上，要提供locale的支持，可能需要用到约300 MB（注9）的文件系统。

注9： MB = megabyte，约1百万字节，一个字节传统上有8位，不过更大或更小的尺寸都有人用过。通常M意即2的20次方，也就是1 048 576。

有些GNU包已完成国际化，并在本地化支持上加入了许多locale。例如，以Italian（意大利文）locale来说，GNU的 `ls` 命令已提供如下的辅助说明：

```
$ LC_ALL=it_IT ls --help              取得GNU ls的Italian辅助说明
Uso: ls [OPZIONE]... [FILE]...
Elenca informazioni sui FILE (predefinito: la directory corrente).
Ordina alfabeticamente le voci se non è usato uno di -cftuSUX oppure --sort.
""
Mandatory arguments to long options are mandatory for short options too.
  -a, --all                  non nasconde le voci che iniziano con .
  -A, --almost-all           non elenca le voci implicite . e ..
      --author               stampa l'autore di ogni file
  -b, --escape               stampa escape ottali per i caratteri non grafici
      --block-size=DIMENS    usa blocchi lunghi DIMENS byte
...
```

注意，没有译文的地方（输出结果的第5行）会回到原本的语言：英文。程序名称及选项名称没有翻译，因为这么做会破坏软件的可移植性。

目前大多数系统均已对国际化与本地化提供些许支持，让Shell程序员得以处理这方面的问题。我们所写的Shell脚本常受到locale的影响，尤其是排序规则（collation order），以及正则表达式（regular expression）的"方括号表示式"（bracket-expression）里的字符范围。不过，当我们在3.2.1节讨论到字符集（character class）、排序符号（collating symbol）与等价字符集（equivalence class）的时候，你会发现，在大多数UNIX系统下，很难从locale文件与工具来判定"字符集与等价字符集"实际上包含了哪些字符，以及有哪些排序符号可用。这也反映出，在目前的系统上，locale的支持仍未成熟。

GNU *gettext* 包（注10）或许可用来支持Shell脚本的国际化与本地化。这个高级主题不在本书的探讨范围，不过相关细节可以在gettext.info在线手册中的"Preparing Shell Scripts for Internationalization"一节找到。

支持locale的系统很多，但缺乏标准的locale名称，因此locale对Shell脚本的可移植性帮助不大，最多只是将 `LC_ALL` 设置为 `C`，强制采用传统的locale。在本书中，当遇到locale的设置可能会产生非预期结果时，我们就会这么做。

2.9 小结

该选编译型语言还是脚本编程语言，通常视应用程序的需求而定。脚本编程语言多半用

注10： 见 *ftp://ftp.gnu.org/gnu/gettext/*。megabyte的简易算法就是把它想成大概一本书的字数（300页×60行/页×60字符/行=1 080 000字符）。

于比编译型语言高级的情况，当你对性能的要求不高，希望尽快开发出程序并以较高级的方式工作时，也就是使用脚本编程语言的好时机。

Shell是UNIX系统中最重要、也是广为使用的脚本语言。因为它的无所不在，而且遵循POSIX标准，这使得写出来的Shell程序多半能够在各厂商的系统下运行。由于Shell函数是一个高级的功能，所有Shell程序其实相当实用，用户只要花一点力气就能做很多事情。

所有的Shell脚本都应该以#!为第一行；这一机制可让你的脚本更有灵活性，你可以选择使用Shell或其他语言来编写脚本。

Shell是一个完整的程序语言。目前，我们已经说明过基本的命令、选项、参数与变量，以及`echo`与`printf`的基本输出。我们也大致介绍了基本的I/O重定向符：<、>、>>以及 |。

Shell会在`$PATH`变量所列举的各个目录中寻找命令。`$PATH`常会包含个人的`bin`目录（用来存储你个人的程序与脚本），你可以在`.profile`文件中将该目录列入到`PATH`里。

我们还看过了取得命令行参数的基本方式，以及简易的执行跟踪。

本章最后讨论的是国际化与本地化。在世界各地的人们对运算需求越来越大的时候，该主题在计算机系统上也日益重要了。对Shell脚本而言，尽管这方面的支持仍然有限，不过Shell程序员还是应该了解locale对他们的程序代码所造成的影响。

第3章

查找与替换

我们在1.2节里曾提及UNIX程序员偏好处理文本的行与列。文本型数据比二进制数据更具灵活性，且UNIX系统也提供许多工具，让用户可以轻松地剪贴文本。

在本章中，我们要讨论的是编写Shell脚本时经常用到的两个基本操作：文本查找（searching）—— 寻找含有特定文本的行，文本替换（substitution）—— 更换找到的文本。

虽然你可以使用简单的固定文本字符串完成很多工作，但是正则表达式（regular expression）能提供功能更强大的标记法，以单个表达式匹配各种实际的文本段。本章会介绍两种由不同的UNIX程序所提供的正则表达式风格，然后再进一步介绍提取文本与重新编排文本的几个重要工具。

3.1 查找文本

以grep程序查找文本（以UNIX的专业术语来说，是匹配文本（matching text））是相当方便的。在POSIX系统上，grep可以在两种正则表达式风格中选择一种，或是执行简单的字符串匹配。

传统上，有三种程序，可以用来查找整个文本文件：

grep

最早的文本匹配程序。使用POSIX定义的基本正则表达式（Basic Regular Expression，BRE），本章稍后会提到这部分。

egrep

扩展式grep（Extended grep）。这个程序使用扩展正则表达式（Extended Regular Expression，ERE）—— 这是一套功能更强大的正则表达式，使用它的代价就是会

耗掉更多的运算资源。在早期出现的PDP-11的机器上，这点事关重大，不过以现在的系统而言，在性能影响上几乎没有太大的差别。

fgrep

快速grep (Fast grep)。这个版本匹配固定字符串而非正则表达式，它使用优化的算法，能更有效地匹配固定字符串。最初的版本，也是唯一可以并行(in parallel)地匹配多个字符串的版本；也就是说，grep与egrep只能匹配单个正则表达式；而fgrep使用不同的算法，却能匹配多个字符串，有效地测试每个输入行里，是否有匹配的查找字符串。

1992 POSIX标准将这三个改版整合成一个grep程序，它的行为是通过不同的选项加以控制。POSIX版本可以匹配多个模式 —— 不管是BRE还是ERE。fgrep与egrep两者还是可用，只是标记为不推荐使用 (deprecated)，即它们有可能在往后的标准里删除。果然，在2001 POSIX标准里，就只纳入合并后的grep命令。不过实际上，egrep与fgrep在所有UNIX与类UNIX的系统上都还是可用的。

3.1.1 简单的grep

grep最简单的用法就是使用固定字符串：

```
$ who                                                  有谁登录了
tolstoy  tty1        Feb 26 10:53
tolstoy  pts/0       Feb 29 10:59
tolstoy  pts/1       Feb 29 10:59
tolstoy  pts/2       Feb 29 11:00
tolstoy  pts/3       Feb 29 11:00
tolstoy  pts/4       Feb 29 11:00
austen   pts/5       Feb 29 15:39 (mansfield-park.example.com)
austen   pts/6       Feb 29 15:39 (mansfield-park.example.com)
$ who | grep -F austen                           austen登录于何处
austen   pts/5       Feb 29 15:39 (mansfield-park.example.com)
austen   pts/6       Feb 29 15:39 (mansfield-park.example.com)
```

范例中使用-F选项，以查找固定字符串**austen**。事实上，只要匹配的模式里未含有正则表达式的meta字符 (metacharacter)，则grep默认行为模式就等同于使用了-F：

```
$ who | grep austen                                  不具 -F，但结果一样
austen   pts/5       Feb 29 15:39 (mansfield-park.example.com)
austen   pts/6       Feb 29 15:39 (mansfield-park.example.com)
```

3.2 正则表达式

本节提供有关正则表达式构造与匹配方式的概述。特别会提及POSIX BRE与ERE构造，因为它们想要将大部分UNIX工具里的两种正则表达式基本风格 (flavors) 加以正式化。

grep

语法

`grep [ options ... ] pattern-spec [ files ... ]`

用途

显示匹配一个或多个模式的文本行。时常会作为管道（pipeline）的第一步，以便对匹配的数据作进一步处理。

主要选项

`-E`

使用扩展正则表达式进行匹配。grep -E可取代传统的egrep。

`-F`

使用固定字符串进行匹配。grep -F可取代传统的fgrep命令。

`-e pat-list`

通常，第一个非选项的参数会指定要匹配的模式。你也可以提供多个模式，只要将它们放在引号里并以换行字符分隔它们。模式以减号开头时，grep会混淆，而将它视为选项。这就是-e选项派上用场的时候，它可以指定其参数为模式——即使它以减号开头。

`-f pat-file`

从*pat-file*文件读取模式作匹配。

`-i`

模式匹配时忽略字母大小写差异。

`-l`

列出匹配模式的文件名称，而不是打印匹配的行。

`-q`

静默地。如果模式匹配匹配，则grep会成功地离开，而不将匹配的行写入标准输出；否则即是不成功。（我们尚未讨论成功/不成功；可参考6.2节）。

`-s`

不显示错误信息。通常与-q并用。

`-v`

显示不匹配模式的行。

行为模式

读取命令行上指名的每个文件。发现匹配查找模式的行时，将它显示来。当指明多个文件时，grep会在每一行前面加上文件名与一个冒号。默认使用BRE。

警告

你可以使用多个-e与-f选项，建立要查找的模式列表。

我们期望你在阅读这本书前，已经接触过正则表达式与文本匹配，并已有些了解。如果是这样，下面的段落将澄清如何使用正则表达式完成具有可移植性的 Shell 脚本。

若你完全没接触过正则表达式，那么这里提到的东西对你来说可能太简略了，你应该先去看看介绍性的资料，例如《Learning the UNIX Operating System》（O'Reilly）或是《sed & awk》（O'Reilly）。因为正则表达式是 UNIX 工具使用和构建模型上的基础，花些时间学习如何使用它们并且好好利用它们，你会不断地从各个层面得到充分的回报。

如果你使用正则表达式处理文本已有多年经验，可能会觉得这里所介绍的内容略嫌粗略。在这种情况下，我们会建议你浏览了第一部分 POSIX BRE 与 ERE 的表格式概括之后，就直接跳到下一节，然后找一些比较深入的资料来阅读，例如《Mastering Regular Expressions》（O'Reilly）。

3.2.1 什么是正则表达式

正则表达式是一种表示方式，让你可以查找匹配特定准则的文本，例如，“以字母 a 开头”。此表示法让你可以写一个表达式，选定或匹配多个数据字符串。

除了传统的 UNIX 正则表达式表示法之外，POSIX 正则表达式还可以做到：

- 编写正则表达式，它表示特定于 locale 的字符序列顺序和等价字符。
- 编写正则表达式，而不必关心系统底层的字符集是什么。

很多的UNIX工具程序沿用某一种正则表达式形式来强化本身的功能。这里列举一部分例子：

- 用来寻找匹配文本行的grep工具族：grep与egrep，以及非标准但很好用的agrep工具（注 1）。
- 用来改变输入流的 sed 流编辑器（stream editor），本章稍后将会介绍。
- 字符串处理程序语言，例如 awk、Icon、Perl、Python、Ruby、Tcl 等。
- 文件查看程序（有时称为分页程序，pagers），例如more、page，与pg，都常出现在商用 UNIX 系统上，另外还有广受欢迎的 less 分页程序（注 2）。

注 1：1992年原始的UNIX版本是在*ftp://ftp.cs.arizona.edu/agrep/agrep-2.04.tar.Z*。Windows 版本则在 *http://www.tgries.de/agrep/337/agrep337.zip*。agrep 不同于我们在本书中介绍的大部分可自由下载的软件，它并不能随意地用于任何目的；你可以参考程序所附的许可文件。

注 2：与 more 对应的双关语。见 *ftp://ftp.gnu.org/gnu/less/*。

- 文本编辑器，例如历史悠久的ed行编辑器、标准的vi屏幕编辑器，还有一些插件（add-on）编辑器，例如emacs、jed、jove、vile、vim等。

正因为正则表达式对于UNIX的使用是这么的重要，所以花些时间把它们弄熟绝对不会错，越早开始就能掌握得越好。

从根本上来看，正则表达式是由两个基本组成部分所建立：一般字符与特殊字符。一般字符指的是任何没有特殊意义的字符，正如下表中所定义的。在某些情况下，特殊字符也可以视为一般字符。特殊字符常称为元字符（metacharacter），本章接下来的部分都会以meta字符表示。表3-1为POSIX BRE与ERE的meta字符列表。

表3-1：POSIX BRE与ERE的meta字符

字符	BRE/ERE	模式含义
\	两者都可	通常用以关闭后续字符的特殊意义。有时则是相反地打开后续字符的特殊意义，例如\(...\)与\{...\}。
.	两者都可	匹配任何单个的字符，但NUL除外。独立程序也可以不允许匹配换行字符。
*	两者都可	匹配在它之前的任何数目（或没有）的单个字符。以ERE而言，此前置字符可以是正则表达式，例如：因为.（点号）表示任一字符，所以.*代表“匹配任一字符的任意长度”。以BRE来说，*若置于正则表达式的第一个字符，不具任何特殊意义。
^	两者都可	匹配紧接着的正则表达式，在行或字符串的起始处。BRE：仅在正则表达式的开头处具此特殊含义，ERE：置于任何位置都具特殊含义。
$	两者都可	匹配前面的正则表达式，在字符串或行结尾处。BRE：仅在正则表达式结尾处具特殊含义。ERE：置于任何位置都具特殊含义。
[...]	两者都可	方括号表达式（bracket expression），匹配方括号内的任一字符。连字符（-）指的是连续字符的范围（注意：范围会因locale而有所不同，因此不具可移植性）。^符号置于方括号里第一个字符则有反向含义：指的是匹配不在列表内（方括号内）的任何字符。作为首字符的一个连字符或是结束方括号（]），则被视为列表的一部分。所有其他的meta字符也为列表的一部分（也就是：根据其字面上的意义）。方括号表达式里可能会含有排序符号（collating symbol）、等价字符集（equivalence class），以及字符集（character class）（文后将有介绍）。

表 3-1：POSIX BRE 与 ERE 的 meta 字符（续）

字符	BRE/ERE	模式含义
\{*n*,*m*\}	BRE	区间表达式（interval expression），匹配在它前面的单个字符重现（occurrences）的次数区间。\{*n*\}指的是重现*n*次；\{*n*,\}则为至少重现（occurrences）*n*次，而\{*n*,*m*\}为重现*n*至*m*次。*n*与*m*的值必须介于0至RE_DUP_MAX（含）之间，后者最小值为255。
\(\)	BRE	将\(与\)间的模式存储在特殊的"保留空间（holding space)"。最多可以将9个独立的子模式（subpattern）存储在单个模式中。匹配于子模式的文本，可通过转义序列（escape sequences）\1至\9，被重复使用在相同模式里。例如**\(ab\).*\1**，指的是匹配于ab组合的两次重现，中间可存在任何数目的字符。
n	BRE	重复在\(与\)方括号内第*n*个子模式至此点的模式。*n*为1至9的数字，1为由左开始。
{*n*,*m*}	ERE	与先前提及BRE的\{*n*,*m*\}一样，只不过方括号前没有反斜杠。
+	ERE	匹配前面正则表达式的一个或多个实例。
?	ERE	匹配前面正则表达式的零个或一个实例。
|	ERE	匹配于|符号前或后的正则表达式。
( )	ERE	匹配于方括号括起来的正则表达式群。

表 3-2 列举了一些简单的范例。

表 3-2：简单的正则表达式匹配范例

表达式	匹配
tolstoy	位于一行上任何位置的7个字母：tolstoy
^tolstoy	7个字母tolstoy，出现在一行的开头
tolstoy$	7个字母tolstoy，出现在一行的结尾
^tolstoy$	正好包括tolstoy这7个字母的一行，没有其他的任何字符
[Tt]olstoy	在一行上的任意位居中，含有Tolstoy或是tolstoy
tol.toy	在一行上的任意位居中，含有tol这3个字母，加上任何一个字符，再接着toy这3个字母
tol.*toy	在一行上的任意位居中，含有tol这3个字母，加上任意的0或多个字符，再继续toy这3个字母（例如，toltoy、tolstoy、tolWHOtoy等）

3.2.1.1 POSIX 方括号表达式

为配合非英语的环境，POSIX标准强化其字符集范围的能力（例如，**[a-z]**），以匹配

非英文字母字符。举例来说，法文的è是字母字符，但以传统字符集**[a-z]**匹配并无该字符。此外，该标准也提供字符序列功能，可用以在匹配及排序字符串数据时，将序列里的字符视为一个独立单位（例如，将locale中ch这两个字符视为一个单位，在匹配与排序时也应这样看待）。越来越广为使用的Unicode字符集标准，进一步地增加了在简单范围内使用它的复杂度，使得它们对于现代应用程序而言更加不适用。

POSIX也在一般术语上作了些变动，我们早先看到的范围表达式在UNIX里通常称为字符集（character class），在POSIX的标准下，现在叫做方括号表达式（bracket expression）。在方括号表达式里，除了字面上的字符（例如z、;等等）之外，另有额外的组成部分，包括：

字符集（Character class）
: 以[:与:]将关键字组合括起来的POSIX字符集。关键字描述各种不同的字符集，例如英文字母字符、控制字符等，见表3-3。

排序符号（Collating symbol）
: 排序符号指的是将多字符序列视为一个单位。它使用[.与.]将字符组合括起来。排序符号在系统所使用的特定locale上各有其定义。

等价字符集（Equivalence class）
: 等价字符集列出的是应视为等值的一组字符，例如e与è。它由取自于locale的名字元素组成，以[=与=]括住。

这三种构造都必须使用方括号表达式。例如**[[:alpha:]!]**匹配任一英文字母字符或惊叹号（!）；而**[[.ch.]]**则匹配于ch（排序元素），但字母c或h则不是。在法文French的locale里，**[[=e=]]**可能匹配于e、è、ë、ê或é。接下来会有字符集、排序符号，以及等价字符集的详细说明。

表3-3描述POSIX字符集。

表3-3：POSIX字符集

类别	匹配字符	类别	匹配字符
[:alnum:]	数字字符	[:lower:]	小写字母字符
[:alpha:]	字母字符	[:print:]	可显示的字符
[:blank:]	空格（space）与定位（tab）字符	[:punct:]	标点符号字符
[:cntrl:]	控制字符	[:space:]	空白（whitespace）字符
[:digit:]	数字字符	[:upper:]	大写字母字符
[:graph:]	非空格（nonspace）字符	[:xdigit:]	十六进制数字

BRE与ERE共享一些常见的特性，不过仍有些重要差异。我们会从BRE的说明开始，再介绍ERE附加的meta字符，最后针对使用相同（或类似）meta字符但拥有不同语义（或含义）的情况进行说明。

3.2.2 基本正则表达式

BRE是由多个组成部分所构建，一开始提供数种匹配单个字符的方式，而后又结合额外的meta字符，进行多字符匹配。

3.2.2.1 匹配单个字符

最先开始是匹配单个字符。可采用集中方式做到：以一般字符、以转义的meta字符、以.（点号）meta字符，或是用方括号表达式：

- 一般字符指的是未列于表3-1的字符，包括所有文字和数字字符、绝大多数的空白（whitespace）字符以及标点符号字符。因此，正则表达式**a**，匹配于字符a。我们可以说，一般字符所表示的就是它们自己，且这种用法应是最直接且易于理解的。所以，**shell**匹配于shell；**WoRd**匹配于WoRd，但不匹配于word。
- 若meta字符不能表示它们自己，那当我们需要让meta字符表示它们自己的时候，该怎么办？答案是转义它。在前面放一个反斜杠来做到这一点。因此，*****匹配于字面上的*，****匹配于字面上的反斜杠，还有**\[**匹配于左方括号（若将反斜杠放在一般字符前，则POSIX标准保留此行为模式为未定义状态。不过通常这种情况下反斜杠会被忽略，只是很少人会这么做）。
- .（点号）字符意即“任一字符”。因此，**a.c**匹配于abc、aac以及aqc。单个点号用以表示自己的情况很少，它多半与其他meta字符搭配使用，这一结合允许匹配多个字符，这部分稍后会提及。
- 最后一种匹配单个字符的方式是使用方括号表达式（bracket expression）。最简单的方括号表达式是直接将字符列表放在方括号里，例如，**[aeiouy]**表示的就是所有小写元音字母。举例来说，**c[aeiouy]t**匹配于cat、cot以及cut（还有cet、cit，与cyt），但不匹配于cbt。

  在方括号表达式里，^放在字首表示是取反（complement）的意思；也就是说，不在方括号列表里的任意字符。所以**[^aeiouy]**指的就是小写元音字符以外的任何字符，例如：大写元音字母、所有辅音字母、数字、标点符号等。

将所有要匹配的字母全列出来是一件无聊又麻烦的事。例如**[0123456789]**指所有数字，或**[0123456789abcdefABCDEF]**表示所有十六进制数字。因此，方括号表达式可以包括字符的范围。像前面提到的两个例子，就可以分别以**[0-9]**与**[0-9a-fA-F]**表示。

警告：一开始，范围表示法匹配字符时，是根据它们在机器中字符集内的数值。因此字符集的不同（ASCII v.s EBCDIC），会使得表示法无法百分之百地具有可移植性，但实际上问题不大，因为几乎所有的UNIX系统都是使用ACSII。

以POSIX的locale来说，某些地方可能会有问题。范围使用的是各个字符在locale排序序列里所定义的位置，与机器字符集里的数值不相关。因此，范围表示法仅在程序运行在locale设置为“POSIX”之下，才具可移植性。前面所提及的POSIX字符集表示法，提供一种可移植方式表示概念，例如“所有数字”，或是“所有字母字符”，因此在方括号表达式内的范围不建议用在新程序里。

在前面的3.2.1节里，我们曾简短地介绍POSIX的排序符号（collating symbol）、等价字符集（equivalence class）以及字符集（character class）。这些是方括号表达式最后出现的组成部分。接下来我们就要说明它们的构造方式。

在部分非英语系的语言里，为了匹配需要，某些成对的字符必须视为单个字符。像这样的成对字符，当它们与语言里的单个字符比较时，都有其排序的定义方式。例如，在Czech与Spanish语系下，ch两个字符会保持连续状态，在匹配时，会视为单个独立单位。

排序（collating）是指给予成组的项目排列顺序的操作。一个POSIX的排序元素由当前locale中的元素名称组成，并由[.与.]括起来。以刚才讨论的ch来说，locale可能会用[.ch.]（我们说“可能”是因为每一个locale都有自己定义的排序元素）。假定[.ch.]是存在的，那么正则表达式**[ab[.ch.]de]**则匹配于字符a、b、d或e，或者是成对的ch；而单独的c或h字符则不匹配。

等价字符集（equivalence class）用来让不同字符在匹配时视为相同字符。等价字符集将类别（class）名称以[=与=]括起来。举例来说，在French的locale下，可能有[=e=]这样的等价字符集，在此类别存在的情况下，正则表达式**[a[=e=]iouy]**就等同于所有小写英文字母元音，以及字母è、é等。

最后一个特殊组成部分：字符集，它表示字符的类别，例如数字、小写与大写字母、标点符号、空白（whitespace）等。这些类别名称定义于[:与:]之间。完整列表如表3-3所示。前POSIX（pre-POSIX）范围表达式对于十进制与十六进制数字的表示（应该）是具有可移植性的，可使用字符集：**[[:digit:]]**与**[[:xdigit:]]**达成。

注意：排序元素、等价字符集以及字符集，都仅在方括号表达式的方括号内认可，也就是说，像**[:alpha:]**这样的正则表达式，匹配字符为a、l、p、h以及:，表示所有英文字母的正确写法应为**[[:alpha:]]**。

在方括号表达式中，所有其他的 meta 字符都会失去其特殊含义。所以`[*\.]`匹配于字面上的星号、反斜杠以及句点。要让]进入该集合，可以将它放在列表的最前面：`[]*\.]`，将]增加至此列表中。要让减号字符进入该集合，也请将它放到列表最前端：`[-*\.]`。若你需要右方括号与减号两者进入列表，请将右方括号放到第一个字符、减号放到最后一个字符：`[]*\.-]`。

最后，POSIX 明确陈述：NUL 字符（数值的零）不需要是可匹配的。这个字符在 C 语言里是用来指出字符串结尾，而POSIX标准则希望让它是直截了当的，通过正规C字符串的使用实现其功能。除此之外，另有其他个别的工具程序不允许使用 .（点号）meta 字符或方括号表达式来进行换行字符匹配。

3.2.2.2 后向引用

BRE 提供一种叫后向引用（backreferences）的机制，指的是“匹配于正则表达式匹配的先前的部分”。使用后向引用的步骤有两个。第一步是将子表达式包围在`\(`与`\)`里；单个模式里可包括至多 9 个子表达式，且可为嵌套结构。

下一步是在同一模式之后使用`\digit`，`digit`指的是介于1至9的数字，指的是“匹配于第 *n* 个先前方括号内子表达式匹配成功的字符”。举例如下：

模式	匹配成功
`\(ab\)\(cd\)[def]*\2\1`	abcdcdab、abcdeeecdab、abcdddeeffcdab、...
`\(why\).*\1`	一行里重现两个 why
`\([[:alpha:]_][[:alnum:]_]*\) = \1;`	简易 C/C++ 赋值语句

后向引用在寻找重复字以及匹配引号时特别好用：

```
\(["']\).*\1        匹配以单引号或双引号括起来的字，例如 'foo' 或 "bar"
```

在这种方法下，就无须担心是单引号或是双引号先找到。

3.2.2.3 单个表达式匹配多字符

匹配多字符最简单的方法就是把它们一个接一个（连接）列出来，所以正则表达式**ab**匹配于ab，..（两个点号）匹配于任意两个字符，而`[[:upper:]][[:lower:]]`则匹配于任意一个大写字符，后面接着任意一个小写字符。不过，将这些字符全列出来只有在简短的正则表达式里才好用。

虽然 .（点号）meta 字符与方括号表达式都提供了一次匹配一个字符的很好方式，但正则表达式真正强而有力的功能，其实是在修饰符（modifier）meta 字符的使用上。这类 meta 字符紧接在具有单个字符的正则表达式之后，且它们会改变正则表达式的含义。

最常用的修饰符为星号（*），表示“匹配0个或多个前面的单个字符”。因此，**ab*c**表示的是“匹配1个**a**、**0**或多个**b**字符以及**a c**”。这个正则表达式匹配的有**ac**、**abc**、**abbc**、**abbbc**等。

注意：“匹配0或多个”不表示“匹配其他的某一个……”，了解这一点是相当重要的。也就是说，正则表达式**ab*c**下，文本aQc是不匹配的——即便是aQc里拥有0个b字符。相对的，以文本ac来说，b*在**ab*c**里表述的是匹配a与c之间含有空字符串（null string）——意即长度为0的字符串（若你先前没遇过字符串长度为0的概念，这里可能得花点时间消化。总之，它在必要的时候会派得上用场，这在本章稍后会有所介绍）。

---

*修饰符是好用的，但它没有限制，你不能用*表示“匹配三个字符，而不是四个字符”，而要使用一个复杂的方括号表达式，表明所需的匹配次数——这也是件很麻烦的事。区间表达式（interval expressions）可以解决这类问题。就像*，它们一样接在单个字符正则表达式后面，控制该字符连续重复几次即为匹配成功。区间表达式是将一个或两个数字，放在\{与\}之间，有3种变化，如下：

\{*n*\}	前置正则表达式所得结果重现*n*次
\{*n*,\}	前置正则表达式所得的结果重现至少*n*次
\{*n*,*m*\}	前置正则表达式所得的结果重现*n*至*m*次

有了区间表达式，要表达像“重现5个a”或是“重现10到42个q”就变得很简单了，这两项分别是：**a\{5\}**与**q\{10,42\}**。

n与***m***的值必须介于0至RE_DUP_MAX（含）之间。RE_DUP_MAX是POSIX定义的符号型常数，且可通过getconf命令取得。RE_DUP_MAX的最小值为255；不过部分系统允许更大值，在我们的GNU/Linux系统中，就遇到很大的值：

```
$ getconf RE_DUP_MAX
32767
```

3.2.2.4 文本匹配锚点

再介绍两个meta字符就能完成整个BRE的介绍了。这两个meta字符是脱字符号（^）与货币符号（$），它们叫做锚点（anchor），因为其用途在限制正则表达式匹配时，针对要被匹配字符串的开始或结尾处进行匹配（^在此处的用法与方括号表达式里的完全不同）。假定现在有一串要进行匹配的字：abcABCdefDEF，我们以表3-4列举匹配的范例。

表 3-4：正则表达式内锚点的范例

模式	是否匹配	匹配文本（粗体）/匹配失败的理由
ABC	是	居中的第4、5及6个字符：abc**ABC**defDEF
^ABC	否	限定匹配字符串的起始处
def	是	居中的第7、8及9个字符：abcABC**def**DEF
def$	否	限制匹配字符串的结尾处
[[:upper:]]\{3\}	是	居中的第4、5及6个字符：abc**ABC**defDEF
[[:upper:]]\{3\}$	是	结尾的第10、11及12个字符：abcDEFdef**DEF**
^[[:alpha:]]\{3\}	是	起始的第1、2及3个字符：**abc**ABCdefDEF

^与$也可同时使用，这种情况是将括起来的正则表达式匹配整个字符串（或行）。有时 **^$** 这样的简易正则表达式也很好用，它可以用来匹配空的（empty）字符串或行列。例如在grep加上-v选项可以用来显示所有不匹配于模式的行，使用上面的做法，便能过滤掉文件里的空（empty）行。

例如，C的源代码在经过处理后，变成了#include文件与#define宏时，这种用法就很有用了，因为这样一来你可以了解C编译器实际上看到的是什么（这是一种初级的调试法，但有时你就是要这么做）。扩展文件（expanded file）里头时常包含的空白或空行通常会比原始码更多，因此要排除空行只要：

```
$ cc -E foo.c | grep -v '^$' > foo.out        预先删除空行
```

^与$仅在BRE的起始与结尾处具有特殊用途。在BRE下，ab^cd里的^表示的，就是自身（^）；同样地，ef$gh里的$在这里表示的也就是字面上的货币字符。它也可能与其他正则表达式连用，例如 **\^** 与 **\$**，或是 **[$]**（注3）。

3.2.2.5 BRE 运算符优先级

在数学表达式里，正则表达式的运算符具有某种已定义的优先级（precedence），指的是某个运算符（优先级较高）将比其他运算符先被处理。表 3-5 提供 BRE 运算符的优先级 —— 由高至低。

表 3-5：BRE 运算符优先级，由高至低

运算符	表示意义
`[..] [==] [::]`	用于字符排序的方括号符号
metacharacter	转义的 meta 字符

注 3：　**[^]** 并非有效的正则表达式。请确认你了解是为什么。

表 3-5：BRE 运算符优先级，由高至低（续）

运算符	表示意义
[]	方括号表达式
\(\) \digit	子表达式与后向引用
* \{ \}	前置单个字符重现的正则表达式
无符号（no symbol）	连续
^ $	锚点（Anchors）

3.2.3 扩展正则表达式

ERE（Extended Regular Expressions）的含义就如同其名字所示：拥有比基本正则表达式更多的功能。BRE 与 ERE 在大多数 meta 字符与功能应用上几乎是完全一致，但 ERE 里有些 meta 字符看起来与 BRE 类似，却具有完全不同的意义。

3.2.3.1 匹配单个字符

在匹配单个字符的情况下，ERE 本质上是与 BRE 是一致的。特别是像一般字符、用以转义 meta 字符的反斜杠，以及方括号表达式，这些行为模式都与先前提及的 BRE 相同。较有名的一个例外出现在 awk 里：其 \ 符号在方括号表达式内表示其他的含义。因此，如需匹配左方括号、连字符、右方括号或是反斜杠，你应该用 [\[\-\]\\]。这是使用上的经验法则。

3.2.3.2 后向引用不存在

ERE 里是没有后向引用的（注 4）。圆括号在 ERE 里具特殊含义，但和 BRE 里的使用又有所不同（这点稍后会介绍）。在 ERE 里，\(与 \) 匹配的是字面上的左括号与右括号。

3.2.2.3 匹配单个表达式与多个正则表达式

ERE 在匹配多字符这方面，与 BRE 有很明显的不同。不过，在 * 的处理上和 BRE 是相同的（注 5）。

注 4： 这在 grep 与 egrep 命令下有不同的影响，这并非正则表达式匹配能力的问题，只是 UNIX 的一种处理方式而已。

注 5： 有一个例外是，* 作为 ERE 的第一个字符是“未定义”的，而在 BRE 中它是指“符合字面的 *”。

区间表达式可用于ERE中，但它们是写在花括号里（{}），且不需前置反斜杠字符。因此我们先前的例子“要刚好重现5个a”以及“重现10个至42个q”，写法分别为：**a{5}**与**q{10,42}**。而\{与\}则可用以匹配字面上的花括号。当在ERE里{找不到匹配的}时，POSIX特意保留其含义为“未定义（undefined）状态”。

ERE另有两个meta字符，可更细腻地处理匹配控制：

?	匹配于0个或一个前置正则表达式
+	匹配于1个或多个前置正则表达式

你可以把?想成是“可选用的”，也就是说，匹配前置正则表达式的文本，要么出现，要么没出现。举例来说，与**ab?c**匹配的有ac与abc，就这两者！（与**ab*c**相较之下，后者匹配于中间有任意个b）。

+字符在概念上与*meta字符类似，不过前置正则表达式要匹配的文本在这里至少得出现一次。因此，**ab+c**匹配于abc、abbc、abbbc，但不匹配于ac。你当然可以把**ab+c**的正则表达式形式换成**abb*c**；无论如何，当前置正则表达式很复杂时，使用+可以少打一点字，当然也减少了打错字的几率！

3.2.3.4 交替

方括号表达式易于表示“匹配于此字符，或其他字符，或…”，但不能指定“匹配于这个序列（sequence），或其他序列（sequence），或…”。要达到后者的目的，你可以使用交替（alternation）运算符，即垂直的一条线，或称为管道字符（|）。你可以简单写好两个字符序列，再以|将其隔开。例如**read|write**匹配于read与write两者、**fast|slow**匹配于fast与slow两者。你还可以使用多个该符号：**sleep|doze|dream|nod off|slumber**匹配于5个表达式。

|字符为ERE运算符里优先级最低的。因此，左边会一路扩展到运算符的左边，一直到一个前置|的字符，或者是到另一个正则表达式的起始。同样地，|的右边也是一路扩展到运算符的右边，一直到后续的|字符，或是到整个正则表达式的结尾。这部分将在下一节探讨。

3.2.3.5 分组

你应该已经发现，在ERE里，我们已提到运算符是被应用到“前置的正则表达式”。这是因为有圆方括号（(...)）提供分组功能，让接下来的运算符可以应用。举例来说，**(why)+**匹配于一个或连续重复的多个why。

在必须用到交替（alternation）时，分组的功能就特别好用了（也是必需的）。它可以让你用以构建复杂并较灵活性的正则表达式。举例来说，**[Tt]he (CPU|computer) is** 指的就是：在 `The`（或 `the`）与 `is` 之间，含有 `CPU` 或 `computer` 的句子。要特别注意的一点是，圆括号在这里是 meta 字符，而非要匹配的输入文本。

将重复（repetition）运算符与交替功能结合时，分组功能也是一定用得到的。**read|write+** 指的是正好一个 `read`，或是一个 `write` 后面接着任意数个 `e` 字符（`writee`、`writeee` 等），比较有用的模式应该是 **(read|write)+**，它指的是：有一个或重现多个 `read`，或者一个或重现多个 `write`。

当然，**(read|write)+** 所指的字符串中间，不允许有空白。**((read|write)[[:space:]]*)+** 的正则表达式看起来虽然比较复杂，不过也比较实际些。乍看之下，这可能会搞不清楚，不过若把这些组成部分分隔开来看，其实就不难理解了。图 3-1 为图解说明。

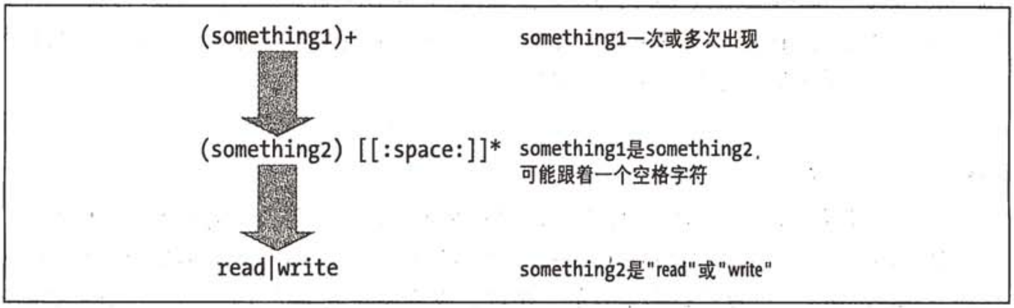


图 3-1：读取一个复杂的正则表达式

结论就是：这个单个正则表达式是用以匹配多个连续出现的 `read` 或是 `write`，且中间可能被空白字符隔开。

在 `[[:space:]]` 之后使用 `*` 是一种判断调用（judgment call）。使用一个 `*` 而非 `+`，此匹配可以取得在行（或字符串）结尾的单词。但也可能可以匹配中间完全没有空白的单词。运用正则表达式时常会需要用到这样的判断调用（judgment call）。该如何构建正则表达式，需要根据输入的数据以及这些数据的用途而定。

最后要说的是：当你将交替（alternation）操作结合 `^` 与 `$` 锚点字符使用时，分组就非常好用了。由于 `|` 为所有运算符中优先级最低的，因此正则表达式 **^abcd|efgh$** 意思是“匹配字符串的起始处是否有 `abcd`，或者字符串结尾处是否有 `efgh`”，这和 **^(abcd|efgh)$** 不一样，后者表示的是“找一个正是 `abcd` 或正是 `efgh` 的字符串”。

3.2.3.6 停驻文本匹配

^ 与 $ 在这里表示的意义与BRE里的相同：将正则表达式停驻在文本字符串（或行）的起始或结尾处。不过有个明显不同的地方就是：在ERE里，^ 与 $ 永远是meta字符。所以，像 **ab^cd** 与 **ef$gh** 这样的正则表达式仍是有效的，只是无法匹配到任何东西，因为 ^ 前置了文本，与 $ 后面的文本，会让它们分别无法匹配到"字符串的开始"与"字符串结尾"。正如其他的meta字符一般，它们在方括号表达式中的确失去了它们特殊的意义。

3.2.3.7 ERE 运算符的优先级

在ERE里运算符的优先级和BRE一样。表3-6由高至低列出了ERE运算符的优先级。

表3-6：ERE运算符优先级，由高至低

运算符	含义
`[..] [= =] [: :]`	用于字符对应的方括号符号
`\metacharacter`	转义的meta字符
`[]`	方括号表达式
`()`	分组
`* + ? {}`	重复前置的正则表达式
无符号（no symbol）	连续字符
`^ $`	锚点（Anchors）
`|`	交替（Alternation）

3.2.4 正则表达式的扩展

很多程序提供正则表达式语法扩展。这类扩展大多采取反斜杠加一个字符，以形成新的运算符。类似POSIX BRE里 `\(...\)` 与 `\{...\}` 的反斜杠。

最常见的扩展为 `\<` 与 `\>` 运算符，分别匹配"单词（word）"的开头与结尾。单词是由字母、数字及下划线组成的。我们称这类字符为单词组成（word-constituent）。

单词的开头要么出现在行起始处，要么出现在第一个后面紧跟一个非单词组成（nonword-constituent）字符的单词组成（word-constituent）字符。同样的，单词的结尾要么出现在一行的结尾处，要么出现在一个非单词组成字符之前的最后一个单词组成字符。

实际上，单词的匹配其实相当直接易懂。正则表达式 **\<chop** 匹配于 `use chopsticks,`

但 eat a lambchop 则不匹配；同样地，**chop\>** 则匹配于第二个字符串，第一个则不匹配。需要特别注意的一点是：在 **\<chop\>** 的表达式下，两个字符串都不匹配。

虽然 POSIX 标准化的只有 ex 编辑器，但在所有商用 UNIX 系统上，ed、ex 以及 vi 编辑器都支持单词匹配，而且几乎已是标准配备。GNU/Linux 与 BSD 系统上附带的克隆程序（"clone" version）也支持单词匹配，还有 emacs、vim 与 vile 也是。除此之外，通常 grep 与 sed 也会支持，不过最好在系统里再通过手册页（manpage）确认一下。

可处理正则表达式的标准工具的 GNU 版本，通常还支持许多额外的运算符，如表 3-7 所示。

表 3-7：额外的 GNU 正则表达式运算符

运算符	含义
\w	匹配任何单词组成字符，等同于 **[[:alnum:]_]**
\W	匹配任何非单词组成字符，等同于 **[^[:alnum:]_]**
\<\>	匹配单词的起始与结尾，如前文所述
\b	匹配单词的起始或结尾处所找到的空字符串。这是 \< 与 \> 运算符的结合 注意：由于 awk 使用 \b 表示后退字符，因此 GNU awk（gawk）使用 \y 表示此功能
\B	匹配两个单词组成字符之间的空字符串
\' \`	分别匹配 emacs 缓冲区的开始与结尾。GNU 程序（还有 emacs）通常将它们视为与 ^ 及 $ 同义

虽然 POSIX 明白表示了 NUL 字符无须是可匹配的，但 GNU 程序则无此限制。若 NUL 字符出现在输入数据里，则它可以通过 .meta 字符或方括号表达式来匹配。

3.2.5 程序与正则表达式

有两种不同的正则表达式风格是经年累月的历史产物。虽然egrep风格的扩展正则表达式在 UNIX 早期开发时就已经存在了，但 Ken Thompson 并不觉得有必要在 ed 编辑器里使用这样全方位的正则表达式（由于 PDP-11 的小型地址空间、扩展正则表达式的复杂度，以及实际应用时大部分的编辑工作使用基本正则表达式已足够了，这样的决定其实相当合理）。

ed 的程序代码后来成了 grep 的基础（grep 为 ed 命令中 g/*re*/p 的缩写，意即全局性匹配 *re*，并将其打印）。ed 的程序代码后来也成为初始构建 sed 的根基。

就在 pre-V7 时期，Al Aho 创造了 egrep，Al Aho 是贝尔实验室的研究人员，他为正则表达式匹配与语言解析的研究奠定了基础。egrep 里的核心匹配程序代码，日后也被 awk 的正则表达式拿来使用。

\< 与 \> 运算符起源于滑铁卢大学的 Rob Pike、Tom Duff、Hugh Redelmeier，以及 David Tilbrook 所修改的 ed 版本（Rob Pike 是这些运算符的发明者之一）。Bill Joy 在 UCB 时，便将这两个运算符纳入 ex 与 vi 编辑器，自那时起，它就广为流传。区间表达式源起于 Programmer's Workbench UNIX（注 6），之后通过 System III 以及此后的 System V，特别将其取出用于商用 UNIX 系统上。表 3-8 列出的是各种 UNIX 程序与其使用的正则表达式。

表 3-8：UNIX 程序及其正则表达式类型

类型	grep	sed	ed	ex/vi	more	egrep	awk	lex
BRE	•	•	•	•	•			
ERE						•	•	•
\< \>	•	•	•	•	•			

lex 是一个很特别的工具，通常是用于语言处理器中的词法分析器的构建。虽然已纳入 POSIX，但我们不会在这里进一步讨论，因为它与 Shell 脚本无关。less 与 pg 虽然不是 POSIX 的一部分，但它们也支持正则表达式。有些系统会有 page 程序，它本质上和 more 是相同的，只是在每个充满屏幕的输出画面之间，会清除屏幕。

正如我们在本章开头所提到的：要（试图）解决多个 grep 的矛盾，POSIX 决定以单个 grep 程序解决。POSIX 的 grep 默认行为模式使用的是 BRE。加上 -E 选项则它使用 ERE，及加上 -F 选项，则它使用 fgrep 的固定字符串匹配算法。因此，真正地遵循 POSIX 的程序应以 grep -E ... 取代 egrep ...。不过，因为所有的 UNIX 系统确实拥有它，且可能已经有许多年了，所以我们继续在自己的脚本中使用它。

最后要注意的一点就是：通常，awk 在其扩展正则表达式里不支持区间表达式。直至 2005 年，各种不同厂商的 awk 版本也并非全面支持区间表达式。为了让程序具有可移植性，若需要在 awk 程序里匹配大方括号，应该以反斜杠转义它，或将它们括在方括号表达式里。

注 6：　Programmer's Workbench (PWB) UNIX 是用在 AT&T 里以支持电信交换软件开发的变化版。它也可以用于商业用途。

3.2.6 在文本文件里进行替换

很多Shell脚本的工作都从通过grep或egrep取出所需的文本开始。正则表达式查找的最初结果，往往就成了要拿来作进一步处理的“原始数据（raw data）”。通常，文本替换（text substitution）至少需要做一件事，就是将一些字以另一些字取代，或者是删除匹配行的某个部分。

一般来说，执行文本替换的正确程序应该是sed——流编辑器（Stream Editor）。sed的设计就是用来以批处理的方式而不是交互的方式来编辑文件。当你知道要做好几个变更——不管是对一个还是数个文件时，比较简单的方式是将这些变更部分写到一个编辑中的脚本里，再将此脚本应用到所有必须修改的文件。sed存在的目的就在这里（虽然你也可以使用ed或ex编辑脚本，但用它们来处理会比较麻烦，而且用户通常不会记得要存储原先的文件）。

我们发现，在Shell脚本里，sed主要用于一些简单的文本替换，所以我们先从它开始。接下来我们还会提供其他的后台数据，并说明sed的功能，特意不在这里提到太多细节，是因为sed所有的功能全都写在《sed & awk》（O'Reilly）这本书里了，该书已列入参考书目。

GNU sed可从*ftp://ftp.gnu.org/gnu/sed/*获取。这个版本拥有相当多有趣的扩展，且已配备使用手册，还附带软件。GNU的sed使用手册里有一些好玩的例子，还包括与众不同的程序测试工具组。可能最令人感到不可思议的是：UNIX dc任意精确度计算程序（arbitrary-precision calculator）竟是以sed所写成的！

当然绝佳的sed来源就是*http://sed.sourceforge.net/*了。这里有连接到两个sed FAQ文件的链接。第一个是*http://www.dreamwvr.com/sed-info/sed-faq.html*，第二个比较旧的FAQ则是*ftp://rtfm.mit.edu/pub/faqs/editor-faq/sed*。

3.2.7 基本用法

你可能会常在管道（pipeline）中间使用sed，以执行替换操作。做法是使用s命令——要求正则表达式寻找，用替代文本（replacement text）替换匹配的文本，以及可选用的标志：

```
sed 's/:.*//' /etc/passwd |           删除第一个冒号之后的所有东西
 sort -u                              排序列表并删除重复部分
```

sed

语法

```
sed [ -n ] 'editing command' [ file ... ]
sed [ -n ] -e 'editing command' ... [ file ... ]
sed [ -n ] -f script-file ... [ file ... ]
```

用途

为了编辑它的输入流，将结果生成到标准输出，而非以交互式编辑器的方式来编辑文件。虽然sed的命令很多，能做很复杂的工作，但它最常用的还是处理输入流的文本替换，通常是作为管道的一部分。

主要选项

-e '*editing command*'

将*editing command*使用在输入数据上。当有多个命令需应用时，就必须使用 -e 了。

-f *script-file*

自*script-file*中读取编辑命令。当有多个命令需要执行时，此选项相当有用。

-n

不是每个最后已修改结果行都正常打印，而是显示以p指定（处理过的）的行。

行为模式

读取每个输入文件的每一行，假如没有文件的话，则是标准输入。以每一行来说，sed会执行每一个应用到输入行的*editing command*。结果会写到标准输出（默认状态下，或是显示地使用p命令及 -n选项）。若无 -e 或 -f 选项，则 sed 会把第一个参数看作是要使用的 *editing command*。

在这里，/字符扮演定界符（delimiter）的角色，从而分隔正则表达式与替代文本（replacement text）。在本例中，替代文本是空的（空字符串 null string），实际上会有效地删除匹配的文本。虽然/是最常用的定界符，但任何可显示的字符都能作为定界符。在处理文件名称时，通常都会以标点符号字符作为定界符（例如分号、冒号或逗点）：

```
find /home/tolstoy -type d -print   |      寻找所有目录
 sed 's;/home/tolstoy/;/home/lt/;'  |      修改名称，注意：这里使用分号作为定界符
  sed 's/^/mkdir /'                 |      插入 mkdir 命令
   sh -x                                   以 Shell 跟踪模式执行
```

上述脚本是将/home/tolstoy目录结构建立一份副本在/home/lt下（可能是为备份而

做的准备）。(find 命令在第 10 章将会介绍，在本例中它的输出是 /home/tolstoy 底下的目录名称列表：一行一个目录。）这个脚本使用了产生命令（generating commands）的手法，使命令内容成为 Shell 的输入。这是一个功能很强且常见的技巧，但却很少人这么用（注 7）。

3.2.7.1 替换细节

先前已经提过，除斜杠外还可以使用其他任意字符作为定界符；在正则表达式或替代文本里，也能转义定界符，不过这么做可能会让命令变得很难看懂：

```
sed 's/\/home\/tolstoy\//\/home\/lt\//'
```

在前面的 3.2.2 节里，我们讲到 POSIX 的 BRE 时，已说明后向引用在正则表达式里的用法。sed 了解后向引用，而且它们还能用于替代文本中，以表示“从这里开始替换成匹配第 *n* 个圆括号里子表达式的文本”：

```
$ echo /home/tolstoy/ | sed 's;\(/home\)/tolstoy/;\1/lt/;'
/home/lt/
```

sed将\1替代为匹配于正则表达式的 **/home** 部分。在这里的例子中，所有的字符都表示它自己，不过，任何正则表达式都可括在 \(与 \) 之间，且后向引用最多可以用到 9 个。

有些其他字符在替代文本里也有特殊含义。我们已经提过需要使用反斜杠转义定界符的情况。当然，反斜杠字符本身也可能需要转义。最后要说明的是：& 在替代文本里表示的意思是“从此点开始替代成匹配于正则表达式的整个文本”。举例来说，假设处理 Atlanta Chamber of Commerce 这串文本，想要在广告册中修改所有对该城市的描述：

```
mv atlga.xml atlga.xml.old
sed 's/Atlanta/&, the capital of the South/' < atlga.xml.old > atlga.xml
```

（作为一个跟得上时代的人，我们在所有的地方都尽可能使用 XML，而不是昂贵的专用字处理程序）。这个脚本会存储一份原始广告小册的备份，做这类操作绝对有必要——特别是还在学习如何处理正则表达式与替换（substitutions）的时候，然后再使用 sed 进行变更。

如果要在替代文本里使用 & 字符的字面意义，请使用反斜杠转义它。例如，下面的小脚本便可以转换 DocBook/XML 文件里字面上的反斜杠，将其转换为 DocBook 里对应的 \：

```
sed 's/\\/\&bsol;/g'
```

注 7：　这个脚本有小瑕疵，它无法处理目录名称含有空格的情况。这个问题是可以解决的，只是要有点小技巧，这部分我们将在第 10 章介绍。

在 s 命令里以 g 结尾表示的是：全局性（global），意即以“替代文本取代正则表达式中每一个匹配的”。如果没有设置 g，sed 只会取代第一个匹配的。这里来比较看看有没有设置 g 所产生的结果：

```
$ echo Tolstoy reads well. Tolstoy writes well. > example.txt   输入样本
$ sed 's/Tolstoy/Camus/' < example.txt                           没有设置 g
Camus reads well. Tolstoy writes well.
$ sed 's/Tolstoy/Camus/g' < example.txt                          设置了 "g"
Camus reads well. Camus writes well.
```

鲜为人知的是（可以用来吓吓朋友）：你可以在结尾指定数字，指示第 *n* 个匹配出现才要被取代：

```
$ sed 's/Tolstoy/Camus/2' < example.txt    仅替代第二个匹配者
Tolstoy reads well. Camus writes well.
```

到目前为止，我们讲的都是一次替换一个。虽然可以将多个 sed 实体以管道（pipeline）串起来，但是给予 sed 多个命令是比较容易的。在命令行上，这是通过 -e 选项的方式来完成。每一个编辑命令都使用一个 -e 选项：

```
sed -e 's/foo/bar/g' -e 's/chicken/cow/g' myfile.xml > myfile2.xml
```

不过，如果你有很多要编辑的项目，这种形式就很恐怖了。所以有时，将编辑命令全放进一个脚本里，再使用 sed 搭配 -f 选项会更好：

```
$ cat fixup.sed
s/foo/bar/g
s/chicken/cow/g
s/draft animal/horse/g
...
$ sed -f fixup.sed myfile.xml > myfile2.xml
```

你也可以构建一个结合 -e 与 -f 选项的脚本；脚本为连续的所有编辑命令，依次提供所有选项。此外，POSIX 也允许使用分号将同一行里的不同命令隔开：

```
sed 's/foo/bar/g ; s/chicken/cow/g' myfile.xml > myfile2.xml
```

不过，许多商用 sed 版本还不支持此功能，所以如果你很在意可移植性的问题，请避免使用此法。

ed 与其先驱 ex 与 vi 一样，sed 会记得在脚本里遇到的最后一个正则表达式 —— 不管它在哪。通过使用空的正则表达式，同一个正则表达式可再使用：

```
s/foo/bar/3        更换第三个 foo
s//quux/           现在更换第一个
```

你可以考虑一个 html2xhtml.sed 的简单脚本，它将 HTML 转换为 XHTML。该脚本会将标签转换成小写，然后更改
 标签为自我结束形式
：

```
s/<H1>/<h1>/g                          斜杠为定界符
s/<H2>/<h2>/g
s/<H3>/<h3>/g
s/<H4>/<h4>/g
s/<H5>/<h5>/g
s/<H6>/<h6>/g
s:</H1>:</h1>:g                        冒号为定界符，因数据内容里已有斜杠
s:</H2>:</h2>:g
s:</H3>:</h3>:g
s:</H4>:</h4>:g
s:</H5>:</h5>:g
s:</H6>:</h6>:g
s:</[Hh][Tt][Mm][LL]>:<html>:g
s:</[Hh][Tt][Mm][Ll]>:</html>:g
s:<[Bb][Rr]>:<br/>:g
...
```

像这样的脚本就可以自动执行大量的HTML转XHTML了，XHTML为标准化的、以XML为主的HTML版本。

3.2.8 sed 的运作

sed的工作方式相当直接。命令行上的每个文件名会依次打开与读取。如果没有文件，则使用标准输入，文件名“-”（单个破折号）可用于表示标准输入。

sed读取每个文件，一次读一行，将读取的行放到内存的一个区域 —— 称之为模式空间（pattern space）。这就像程序语言里的变量一样：内存的一个区域在编辑命令的指示下可以修改，所有编辑上的操作都会应用到模式空间的内容。当所有操作完成后，sed会将模式空间的最后内容打印到标准输出，再回到开始处，读取另一个输入行。

这一工作过程如图3-2所示。脚本使用两条命令，将The UNIX System替代为The UNIX Operating System。

3.2.8.1 打印与否

-n选项修改了sed的默认行为。当提供此选项时，sed将不会在操作完成后打印模式空间的最后内容。反之，若在脚本里使用p，则会明白地将此行显示出来。举例来说，我们可以这样模拟grep：

```
sed -n '/<HTML>/p' *.html          仅显示<HTML>这行
```

虽然这个例子很简单，但这个功能在复杂的脚本里非常好用。如果你使用一个脚本文件，可通过特殊的首行来打开此功能：

```
#n                                 关闭自动打印
/<HTML>/p                          仅打印含<HTML>的行
```

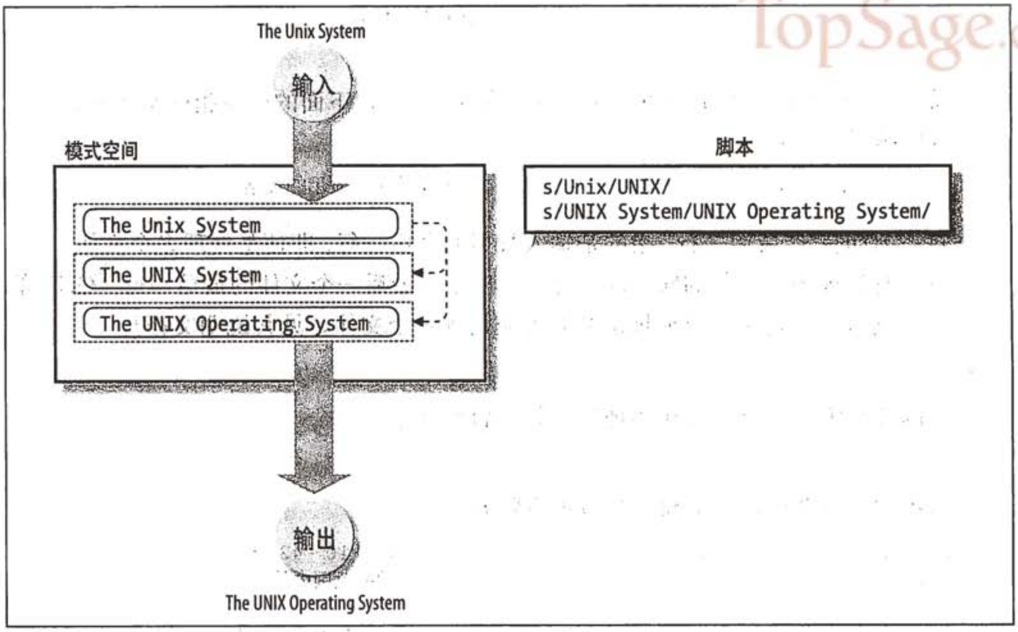


图 3-2：在 sed 脚本中的命令改变了模式空间

在 Shell 中，与很多其他 UNIX 脚本式语言一样：# 是注释的意思。sed 注释必须出现在单独的行里，因为它们是语法型命令，意思是：它们是什么事也不做的命令。虽然POSIX指出，注释可以放在脚本里的任何位置，但很多旧版sed仅允许出现在首行，GNU sed 则无此限制。

3.2.9 匹配特定行

如前所述，sed 默认地会将每一个编辑命令（editing command）应用到每个输入行。而现在我们要告诉你的是：还可以限制一条命令要应用到哪些行，只要在命令前置一个地址（address）即可。因此，sed 命令的完整形式就是：

```
address command
```

以下为不同种类的地址：

正则表达式

将一模式放置到一条命令之前，可限制命令应用于匹配模式的行。可与 s 命令搭配使用：

```
/oldfunc/ s/$/# XXX: migrate to newfunc/      注释部分源代码
```

s 命令里的空模式指的是“使用前一个正则表达式”：

```
/Tolstoy/ s//& and Camus/g              提及两位作者
```

最终行

符号 $（就像在 ed 与 ex 里一样）指“最后一行”。下面的脚本指的是快速打印文件的最后一行：

```
sed -n '$p' "$1"                        引号里为指定显示的数据
```

对 sed 而言，“最后一行”指的是输入数据的最后一行。即便是处理多个文件，sed 也将它们视为一个长的输入流，且 $ 只应用到最后一个文件的最后一行（GNU 的 sed 具有一个选项，可使地址分开地应用到每个文件，见其说明文档）。

行编号

可以使用绝对的行编号作为地址。稍后将有介绍。

范围

可指定行的范围，仅需将地址以逗点隔开：

```
sed -n '10,42p' foo.xml                 仅打印 10～42 行
sed '/foo/,/bar/ s/baz/quux/g'          仅替换范围内的行
```

第二个命令为“从含有 **foo** 的行开始，再匹配是否有 **bar** 的行，再将匹配后的结果中，有 **baz** 的全换成 quux”（像 ed、ex 这类的检阅程序，或是 vi 内的冒号命令提示模式下，都认识此语法）。

这种以逗点隔开两个正则表达式的方式称为范围表达式（range expression）。在 sed 里，总是需要使用至少两行才能表达。

否定正则表达式

有时，将命令应用于不匹配于特定模式的每一行，也是很有用的。通过将!加在正则表达式后面就能做到，如下所示：

```
/used/!s/new/used/g                     将没有 used 的每个行里所有的 new 改成 used
```

POSIX 标准指出：空白（whitespace）跟随在！之后的行为是“未定义的（unspecified）”，并建议需提供完整可移植性的应用软件，不要在！之后放置任何空白字符，这明显是由于某些 sed 的古董级版本仍无法识别它。

例 3-1 说明的是使用绝对的行编号作为地址的用法，这里是以 sed 展现的 head 程序简易版。

例 3-1：使用 sed 的 head 命令

```
# head --- 打印前 n 行
#
# 语法:   head N file

count=$1
sed ${count}q "$2"
```

当你引用head 10 foo.xml之后，sed会转换成sed 10q foo.xml。q命令要求sed马上离开；不再读取其他输入，或是执行任何命令。后面的7.6.1节里，我们将介绍如何让这个脚本看起来更像真正的head命令。

迄今为止，我们看到的都是sed以/字符隔开模式以便查找。在这里，我们要告诉你如何在模式内使用不同的定界符：这通过在字符前面加上一个反斜杠实现：

```
$ grep tolstoy /etc/passwd                                显示原始行
tolstoy:x:2076:10:Leo Tolstoy:/home/tolstoy:/bin/bash
$ sed -n '\:tolstoy: s;;Tolstoy;p' /etc/passwd            改变定界符
Tolstoy:x:2076:10:Leo Tolstoy:/home/tolstoy:/bin/bash
```

本例中，以冒号隔开要查找的模式，而分号则扮演s命令的定界符角色（编辑上的操作其实不重要，我们的重点是介绍如何使用不同的定界符）。

3.2.10 有多少文本会改动

有个问题我们一直还没讨论到：有多少文本会匹配?事实上，这应该包含两个问题。第二个问题是：从哪儿开始匹配？执行简单的文本查找，例如使用grep或egrep时，则这两个问题都不重要，你只要知道是否有一行是匹配的，若有，则看看那一行是什么。至于在这个行里，是从哪儿开始匹配，或者它扩展到哪里，已经不重要了。

但如果你要使用sed执行文本替换，或者要用awk写程序，这两个问题的答案就变得非常重要了（如果你每天都在文本编辑器内工作，这也算是个重要议题，只是本书的重点不在文本编辑器）。

这两个问题的答案是：正则表达式匹配可以匹配整个表达式的输入文本中最长的、最左边的子字符串。除此之外，匹配的空（null）字符串，则被认为是比完全不匹配的还长（因此，就像我们先前所解释的，正则表达式：**ab*c** 匹配文本ac，而b*则成功地匹配于a与c之间的null字符串）。再者，POSIX标准指出："完全一致的匹配，指的是自最左边开始匹配、针对每一个子模式、由左至右，必须匹配到最长的可能字符串"。（子模式指的是在ERE下圆括号里的部分。为此目的，GNU的程序通常也会在BRE里以\(...\)提供此功能）。

如果sed要替代由正则表达式匹配的文本，那么确定该正则表达式匹配的字不会太少或太多就非常重要了。这里有个简单例子：

```
$ echo Tolstoy writes well | sed 's/Tolstoy/Camus/'     使用固定字符串
Camus writes well
```

当然，sed可以使用完整的正则表达式。这里就是要告诉你，了解"从最长的最左边（longest leftmost）"规则有多的重要：

```
$ echo Tolstoy is worldly | sed 's/T.*y/Camus/'      试试正则表达式
Camus                                                结果呢 ?
```

很明显，这里只是要匹配Tolstoy，但由于匹配会扩展到可能的最长长度的文本量，所以就一直找到worldly的y了！你需要定义的更精确些：

```
$ echo Tolstoy is worldly | sed 's/T[[:alpha:]]*y/Camus/'
Camus is worldly
```

通常，当开发的脚本是要执行大量文本剪贴和排列组合时，你会希望谨慎地测试每样东西，确认每个步骤都是你要的 —— 尤其是当你还在学习正则表达式里的微妙变化的阶段的时候。

最后，正如我们所见到的，在文本查找时有可能会匹配到null字符串。而在执行文本替代时，也允许你插入文本：

```
$ echo abc | sed 's/b*/1/'        替代第一个匹配成功的
1abc
$ echo abc | sed 's/b*/1/g'       替代所有匹配成功的
1a1c1
```

请留意，**b*** 是如何匹配在abc的前面与结尾的null字符串。

3.2.11 行v.s.字符串

了解行（line）与字符串（string）的差异是相当重要的。大部分简易程序都是处理输入数据的行，像grep与egrep，以及sed大部分的工作（99%）都是这样。在这些情况下，不会有内嵌的换行字符出现在将要匹配的数据中，^与$则分别表示行的开头与结尾。

然而，对可应用正则表达式的程序语言，例如awk、Perl以及Python，所处理的就多半是字符串。若每个字符串表示的就是独立的一行输入，则^与$仍旧可分别表示行的开头与结尾。不过这些程序语言，其实可以让你使用不同的方式来标明每条输入记录的定界符，所以有可能单独的输入“行”（记录）里会有内嵌的换行字符。这种情况下，^与$无法匹配内嵌的换行字符；它们只用来表示字符串的开头与结尾。当你开始使用可程序化的软件工具时，这一点，请牢记在心。

3.3 字段处理

很多的应用程序，会将数据视为记录与字段的结合，以便于处理。一条记录（record）指的是相关信息的单个集合，例如以企业来说，记录可能含有顾客、供应商以及员工等数据，以学校机构来说，则可能有学生数据。而字段（field）指的就是记录的组成部分，例如姓、名或者街道地址。

3.3.1 文本文件惯例

由于UNIX鼓励使用文本型数据，因此系统上最常见的数据存储类型就是文本了，在文本文件下，一行表示一条记录。这里要介绍的是在一行内用来分隔字段的两种惯例。首先是直接使用空白（whitespace），也就是用空格键（space）或制表（tab）键：

```
$ cat myapp.data
# model      units sold      salesperson
xj11         23              jane
rj45         12              joe
cat6         65              chris
...
```

本例中，#字符起始的行表示注释，可忽略（这是一般的习惯，注释行的功能相当好用，不过软件必须可忽略这样的行才行）。各字段都以任意长度的空格（space）或制表（Tab）字符隔开。第二种惯例是使用特定的定界符来分隔字段，例如冒号：

```
$ cat myapp.data
# model:units sold:salesperson
xj11:23:jane
rj45:12:joe
cat6:65:chris
...
```

两种惯例都有其优缺点。使用空白作为分隔时，字段内容就最好不要有空白（若你使用制表字符（Tab）作分隔，字段里有空格是不会有问题的，但这么做视觉上会混淆，因为你在看文件时，很难马上分辨出它们的不同）。反过来说，若你使用显式的定界符，那么该定界符也最好不要成为数据内容。请你尽可能小心地选择定界符，让定界符出现在数据内容里的可能性降到最低或不存在。

注意： 这两种方式最明显的不同，便是在处理多个连续重复的定界符之时。使用空白(whitespace)分隔时，通常多个连续出现的空格或制表字符都将看作一个定界符。然而，若使用的是特殊字符分隔，则每个定界符都隔开一个字段，例如，在myapp.data的第二个版本里使用的两个冒号字符（“::”）则会分隔出一个空的字段。

以定界符分隔字段最好的例子就是/etc/passwd了，在这个文件里，一行表示系统里的一个用户，每个字段都以冒号隔开。在本书中，很多地方都会以/etc/passwd为例，因为在系统管理工作中，很多时候都是在处理这个文件。如下是该文件的典型例子：

```
tolstoy:x:2076:10:Leo Tolstoy:/home/tolstoy:/bin/bash
```

该文件含有7个字段，分别是：

1. 用户名称
2. 加密后的密码（如账号为停用状态，此处为一个星号，或是若加密后的密码文件存储于另外的 /etc/shadow 里，则这里也可能是其他字符）。
3. 用户 ID 编号。
4. 用户组 ID 编号。
5. 用户的姓名，有时会另附其他相关数据（办公室号码、电话等）。
6. 根目录。
7. 登录的 Shell。

某些UNIX工具在处理以空白界定字段的文件时，做得比较好，有些则是以定界符分隔字段比较好，更有其他的工具两种方式都能处理得当，这部分我们稍后会介绍。

3.3.2 使用 cut 选定字段

cut 命令是用来剪下文本文件里的数据，文本文件可以是字段类型或是字符类型。后一种数据类型在遇到需要从文件里剪下特定的列时，特别方便。请注意：一个制表字符在此被视为单个字符（注 8）。

举例来说，下面的命令可显示系统上每个用户的登录名称及其全名：

```
$ cut -d : -f 1,5 /etc/passwd          取出字段
root:root                              管理者账号
...
tolstoy:Leo Tolstoy                    实际用户
austen:Jane Austen
camus:Albert Camus
...
```

通过选择其他字段编号，还可以取出每个用户的根目录：

```
$ cut -d : -f 6 /etc/passwd            取出根目录
/root                                  管理账号
...
/home/tolstoy                          实际用户
/home/austen
/home/camus
...
```

通过字符列表做剪下操作有时是很方便的。例如，你只要取出命令 ls -l 的输出结果中的文件权限字段：

注 8： 这可通过 expand 与 unexpand 改变其定义。见 *expand*(1)手册页。

Cut

语法

```
cut -c list [ file ... ]
cut -f list [ -d delim ] [ file ... ]
```

用途

从输入文件中选择一或多个字段或者一组字符，配合管道（pipeline），可再做进一步处理。

主要选项

`-c list`

以字符为主，执行剪下的操作。*list* 为字符编号或一段范围的列表（以逗点隔开），例如 1,3,5-12,42。

`-d delim`

通过 -f 选项，使用 *delim* 作为定界符。默认的定界符为制表字符（Tab）。

`-f list`

以字段为主，作剪下的操作。*list* 为字段编号或一段范围的列表（以逗点隔开）。

行为模式

剪下输入字符中指定的字段或指定的范围。若处理的是字段，则定界符隔开的即为各字段，而输出时字段也以给定的定界符隔开。若命令行没有指定文件，则读取标准输入。见正文中的范例。

警告

于 POSIX 系统下，cut 识别多字节字符。因此，"字符（character）"与"字节（byte）"意义不同。详细内容见 *cut*(1) 的手册页。

有些系统对输入行的大小有所限制，尤其是含有多字节字符（multibyte characters）时，这点请特别留意。

```
$ ls -l | cut -c 1-10
total 2878
-rw-r--r--
drwxr-xr-x
-r--r--r--
-rw-r--r--
...
```

不过这种用法比使用字段的风险要大。因为你无法保证行内的每个字段长度总是一样的。一般来说，我们偏好以字段为基础来提取数据。

3.3.3 使用 join 连接字段

join命令可以将多个文件结合在一起，每个文件里的每条记录，都共享一个键值(key)，键值指的是记录中的主字段，通常会是用户名称、个人姓氏、员工编号之类的数据。举例来说，有两个文件，一个列出所有业务员销售业绩，一个列出每个业务员应实现的业绩：

join

语法

    join [options ...] file1 file2

用途

以共同一个键值，将已存储文件内的记录加以结合。

主要选项

-1 *field1*

-2 *field2*

标明要结合的字段。-1 *field1* 指的是从 *file1* 取出 *field1*，而 -2 *field2* 指的则为从 *file2* 取出 *field2*。字段编号自 1 开始，而非 0。

-o *file.field*

输出 *file* 文件中的 *field* 字段。一般的字段则不打印。除非使用多个 -o 选项，即可显示多个输出字段。

-t *separator*

使用 *separator* 作为输入字段分隔字符，而非使用空白。此字符也为输出的字段分隔字符。

行为模式

读取 *file1* 与 *file2*，并根据共同键值结合多笔记录。默认以空白分隔字段。输出结果则包括共同键值、来自 *file1* 的其余记录，接着 *file2* 的其余记录（指除了键值外的记录）。若 *file1* 为 -，则 join 会读取标准输入。每个文件的第一个字段是用来结合的默认键值；可以使用 -1 与 -2 更改之。默认情况下，在两个文件中未含键值的行将不打印（已有选项可改变，见 *join*(1)手册页）。

警告

-1 与 -2 选项的用法是较新的。在较旧的系统上，可能得用：-j1 *field1* 与 -j2 *field2*。

```
$ cat sales                            显示sales 文件
# 业务员数据                           注释说明
# 业务员 量
joe       100
jane      200
herman    150
chris     300
```

```
$ cat quotas                           显示quotas 文件
# 配额
# 业务员 配额
joe       50
jane      75
herman    80
chris     95
```

每条记录都有两个字段：业务员的名字与相对应的量。在本例中，列与列之间有多个空白，从而可以排列整齐。

为了让join运作得到正确结果，输入文件必须先完成排序。例3-2里的程序merge-sales.sh即为使用join结合两个文件。

例 3-2：merge-sales.sh

```
#! /bin/sh

# merge-sales.sh
#
# 结合配额与业务员数据

# 删除注释并排序数据文件
sed '/^#/d' quotas | sort > quotas.sorted
sed '/^#/d' sales  | sort > sales.sorted

# 以第一个键值作结合，将结果产生至标准输出
join quotas.sorted sales.sorted

# 删除缓存文件
rm quotas.sorted sales.sorted
```

首先，使用sed删除注释，然后再排序个别文件。排序后的缓存文件成为join命令的输入数据，最后删除缓存文件。这是执行后的结果：

```
$ ./merge-sales.sh
chris 95 300
herman 80 150
jane 75 200
joe 50 100
```

3.3.4 使用 awk 重新编排字段

awk 本身所提供的功能完备，已经是一个很好用的程序语言了。我们在第9章会好好地介绍该语言的精髓。虽然 awk 能做的事很多，但它主要的设计是要在 Shell 脚本中发挥所长：做一些简易的文本处理，例如取出字段并重新编排这一类。本节，我们将介绍 awk 的基本概念，随后你看到这样的“单命令行程序（one-liners）”就会比较了解了。

3.3.4.1 模式与操作

awk 的基本模式不同于绝大多数的程序语言。它其实比较类似于 sed：

```
awk 'program' [ file ... ]
```

awk 读取命令行上所指定的各个文件（若无，则为标准输入），一次读取一条记录（行）。再针对每一行，应用程序所指定的命令。awk 程序基本架构为：

```
pattern { action }
pattern { action }
...
```

*pattern*部分几乎可以是任何表达式，但是在单命令行程序里，它通常是由斜杠括起来的 ERE。*action* 为任意的 awk 语句，但是在单命令行程序里，通常是一个直接明了的 print 语句（稍后有范例说明）。

pattern 或是 *action* 都能省略（当然，你不会两个都省略吧？）。省略 *pattern*，则会对每一条输入记录执行 *action*；省略 *action* 则等同于{ print }，将打显示整条记录（稍后将会介绍）。大部分单命令行程序为这样的形式：

```
... | awk '{ print some-stuff }' | ...
```

对每条记录来说，awk 会测试程序里的每个 *pattern*。若模式值为真（例如某条记录匹配于某正则表达式，或是一般表达式计算为真），则 awk 便执行 *action* 内的程序代码。

3.3.4.2 字段

awk 设计的重点就在字段与记录上：awk 读取输入记录（通常是一些行），然后自动将各个记录切分为字段。awk 将每条记录内的字段数目，存储到内建变量 NF。

默认以空白分隔字段——例如空格与制表字符（或两者混用），像 join 那样。这通常就足够使用了，不过，其实还有其他选择：你可以将 FS 变量设置为一个不同的值，也就可以变更 awk 分隔字段的方式。如使用单个字符，则该字符出现一次，即分隔出一个字段（像 cut -d 那样）。或者，awk 特别之处就是：也可以设置它为一个完整的 ERE，这种情况下，每一个匹配在该 ERE 的文本都将视为字段分隔字符。

如需字段值，则是搭配 $ 字符。通常 $ 之后会接着一个数值常数，也可能是接着一个表达式，不过多半是使用变量名称。列举几个例子如下：

`awk '{ print $1 }'`	打印第 1 个字段（未指定 pattern）
`awk '{ print $2, $5 }'`	打印第 2 与第 5 个字段（未指定 pattern）
`awk '{ print $1, $NF }'`	打印第 1 个与最后一个字段（未指定 pattern）
`awk 'NF > 0 { print $0 }'`	打印非空行（指定 pattern 与 action）
`awk 'NF > 0'`	同上（未指定 action，则默认为打印）

比较特别的字段是编号 0：表示整条记录。

3.3.4.3 设置字段分隔字符

在一些简单程序中，你可以使用 -F 选项修改字段分隔字符。例如，显示 /etc/passwd 文件里的用户名称与全名，你可以：

```
$ awk -F: '{ print $1, $5 }' /etc/passwd        处理 /etc/passwd
root root                                       管理账号
...
tolstoy Leo Tolstoy                             实际用户
austen Jane Austen
camus Albert Camus
...
```

-F 选项会自动地设置 FS 变量。请注意，程序不必直接参照 FS 变量，也不用必须管理读取的记录并将它们分割为字段：awk 会自动完成这些事。

你可能已经发现，每个输出字段是以一个空格来分隔的 —— 即便是输入字段的分隔字符为冒号。awk 的输入、输出分隔字符用法是分开的，这点与其他工具程序不同。也就是说，必须设置 OFS 变量，改变输出字段分隔字符。方式是在命令行里使用 -v 选项，这会设置 awk 的变量。其值可以是任意的字符串。例如：

```
$ awk -F: -v 'OFS=**'  '{ print $1, $5 }' /etc/passwd     处理 /etc/passwd
root**root                                                管理者账号
...
tolstoy**Leo Tolstoy                                      实际用户
austen**Jane Austen
camus**Albert Camus
...
```

稍后就可以看到设置这些变量的其他方式。或许那些方式更易于理解，根据你的喜好而定。

3.3.4.4 打印行

就像我们已经所介绍过的：大多数时候，你只是想把选定的字段显示来，或者重新安排其顺序。简单的打印可使用print语句做到，只要提供给它需要打印的字段列表、变量或字符串即可：

```
$ awk -F: '{ print "User", $1, "is really", $5 }' /etc/passwd
User root is really root
...
User tolstoy is really Leo Tolstoy
User austen is really Jane Austen
User camus is really Albert Camus
...
```

简单明了的print语句，如果没有任何参数，则等同于print $0，即显示整条记录。

以刚才的例子来说，在混合文本与数值的情况下，多半会使用awk版本的printf语句。这和先前在2.5.4节所提及的Shell（与C）版本的printf语句相当类似，这里就不再重复。以下是把上例修改为使用printf语句的用法：

```
$ awk -F: '{ printf "User %s is really %s\n", $1, $5 }' /etc/passwd
User root is really root
...
User tolstoy is really Leo Tolstoy
User austen is really Jane Austen
User camus is really Albert Camus
...
```

awk的print语句会自动提供最后的换行字符，就像Shell层级的echo与printf那样，然而，如果使用printf语句，则用户必须要通过\n转义序列的使用自己提供它。

注意： 请记得在print的参数间用逗点隔开！否则，awk将连接相邻的所有值：

```
$ awk -F: '{ print "User" $1 "is really" $5 }' /etc/passwd
Userrootis reallyroot
...
Usertolstoyis reallyLeo Tolstoy
Useraustenis reallyJane Austen
Usercamusis reallyAlbert Camus
...
```

这样将所有字符串连在一起应该不是你要的。忘了加上逗点，这是个常见又难找到的错误。

3.3.4.5 起始与清除

BEGIN与END这两个特殊的“模式”，它们提供awk程序起始（startup）与清除（cleanup）操作。常见于大型awk程序中，且通常写在个别文件里，而不是在命令行上：

```
BEGIN       { 起始操作程序代码(startup code) }
pattern1    { action1 }
pattern2    { action2 }
END         { 清除操作程序代码(cleanup code) }
```

BEGIN与END的语句块是可选用的。如需设置，习惯上（但不必须）它们应分别置于awk程序的开头与结尾处。你可以有数个BEGIN与END语句块，awk会按照它们出现在程序的顺序来执行：所有的BEGIN语句块都应该放在起始处，而所有END语句块也应放在结尾。以简单程序来看，BEGIN可用来设置变量：

```
$ awk 'BEGIN { FS = ":" ; OFS = "**" }        使用BEGIN设置变量
> { print $1, $5 }' /etc/passwd               被引用的程序继续到第二行
root**root
...
tolstoy**Leo Tolstoy                          输出，如前
austen**Jane Austen
camus**Albert Camus
...
```

警告：POSIX标准中描述了awk语言及其程序选项。POSIX awk是构建在所谓的"新awk"上，首度全球发布是在1987年的System V Release 3.1版，且在1989年的System V Release 4版中稍作修正。

但是，直到2005年底，Solaris的/bin/awk仍然还是原始的、1979年的awk V7版！在Solaris系统上，你应该使用/usr/xpg4/bin/awk，或参考第9章，使用awk自由下载版中的一个。

3.4 小结

如需从输入的数据文件中取出特定的文本行，主要的工具为grep程序。POSIX采用三种不同grep变体：grep、egrep与fgrep的功能，整合为单个版本，通过不同的选项，分别提供这三种行为模式。

虽然你可以直接查找字符串常数，但是正则表达式能提供一个更强大的方式，描述你要找的文本。大部分的字符在匹配时，表示的是自己本身，但有部分其他字符扮演的是meta字符的角色，也就是指定操作，例如"匹配0至多个的……"、"匹配正好10个的……"等。

POSIX的正则表达式有两种：基本正则表达式（BRE）以及扩展正则表达式（ERE）。哪个程序使用哪种正则表达式风格，是根据长时间的实际经验，由POSIX制定规格，简化

到只剩两种正则表达式的风格。通常，ERE比BRE功能更强大，不过不见得任何情况下都是这样。

正则表达式对于程序执行时的locale环境相当敏感；方括号表达式里的范围应避免使用，改用字符集，例如`[[:alnum:]]`较佳。另外，许多GNU程序都有额外的meta字符。

`sed`是处理简单字符串替换（substitution）的主要工具。在我们的经验里，大部分的Shell脚本在使用`sed`时几乎都是用来作替换的操作，我们特意在这里不介绍`sed`所能提供的其他任务，是因为已经有《sed & awk》这本书（已列于参考书目中），它会介绍更多相关信息。

"从最左边开始，扩展至最长（longest leftmost）"，这个法则描述了匹配的文本在何处匹配以及匹配扩展到多长。在使用`sed`、`awk`或其他交互式文本编辑程序时，这个法则相当重要。除此之外，一行与一个字符串之间的差异也是核心观念。在某些程序语言里，单个字符串可能包含数行，那种情况下，`^`与`$`指的分别是字符串的开头与结尾。

很多时候，在操作上可以将文本文件里的每一行视为一条单个记录，而在行内的数据则包括字段。字段可以被空白或是特殊定界符分隔，且有许多不同的UNIX工具可处理这两种数据。`cut`命令用以剪下选定的字符范围或字段，`join`则是用来结合记录中具有共同键值的字段的文件。

`awk`多半用于简单的"单命令行程序"，当你想要只显示选定的字段，或是重新安排行内的字段顺序时，就是`awk`派上用场的时候了。由于它是编程语言，即使是在简短的程序里，它也能发挥其强大的功能、灵活性与控制能力。

第4章

文本处理工具

有些在文本文件上的操作，之所以能成为广泛应用的标准工具，是因为这些工作早在贝尔实验室里使用UNIX时就开发了。在本章中，我们就是要来看看这些重要工具。

4.1 排序文本

含有独立数据记录的文本文件，通常都可以拿来排序。一个可预期的记录次序，会让用户的生活更便利：书的索引、字典、目录以及电话簿，如果没有次序依据就毫无价值。排序后的记录更易于程序化，也更有效率，这部分在第5章将有进一步的说明。

就像awk、cut与join一样：sort将输入看作具有多条记录的数据流，而记录是由可变宽度的字段组成，记录是以换行字符作为定界符，字段的定界符则是空白字符或是用户指定的单个字符。

4.1.1 行的排序

以最简单的情况来说，未提供命令行选项时，整个记录都会根据当前locale所定义的次序排序。在传统的C locale中，也就是ASCII顺序，但是你可以像我们先前介绍过的2.8节那样，自行设置另一种locale。

在ISO 8859-1小型双语字典里，有4个法文单词，它们的不同之处仅在于重音位置不同：

```
$ cat french-english                    显示迷你字典
côte    coast
cote    dimension
coté    dimensioned
côté    side
```

sort

语法

```
sort [ options ] [ file(s) ]
```

用途

将输入行按照键值字段与数据类型选项以及locale排序。

主要选项

-b

忽略开头的空白。

-c

检查输入是否已正确地排序。如输入未经排序，但退出码（exit code）为非零值，则不会有任何输出。

-d

字典顺序：仅文字数字与空白才有意义。

-g

一般数值：以浮点数字类型比较字段。这个选项的运作有点类似-n，差别仅在于这个选项的数字可能有小数点及指数（例：6.022e+23）。仅GNU版本提供此功能。

-f

将混用的字母都看作相同大小写，也就是以不管字母大小写的方式排序。

-i

忽略无法打印的字符。

-k

定义排序键值字段。详见4.1.2节。

-m

将已排序的输入文件，合并为一个排序后的输出数据流。

-n

以整数类型比较字段。

-o *outfile*

将输出写到指定的文件，而非标准输出。如果该文件为输入文件之一，则sort在进行排序与写到输出文件之前，会先将它复制到一个临时的文件。

-r

倒置排序的顺序为由大至小（descending），而非默认的由小至大（ascending）。

sort（续）

-t *char*

使用单个字符 *char* 作为默认的字段分隔字符，取代默认的空白字符。

-u

只有唯一的记录：丢弃所有具相同键值的记录，只留其中的第一条。只有键值字段是重要的，也就是说：被丢弃的记录其他部分可能是不同值。

行为模式

sort 会读取指定的文件，如果未给定文件，则读取标准输入，再将排序好的数据写至标准输出。

要了解排序的过程，你可以使用八进制的打印工具：od，用ASCII和八进制码来显示法文单词：

```
$ cut -f1 french-english | od -a -b        以八进制字节显示法文单词
0000000   c   t   t   e  nl   c   o   t   e  nl   c   o   t   i  nl   c
        143 364 164 145 012 143 157 164 145 012 143 157 164 351 012 143
0000020   t   t   i  nl
        364 164 351 012
0000024
```

显然，因为加了ASCII选项 -a，od 脚本去掉了字符前面的位，因此重音字母已被切除，不过我们还是可以看到它们的八进制值：é 为 351_8 而 ô 为 364_8。

在GNU/Linux 系统下，可以用如下方式来确认字符值：

```
$ man iso_8859_1                        查看ISO 8859-1的手册页
...
        Oct   Dec   Hex   Char   Description
        --------------------------------------------------------------------
...
        351   233   E9    é      LATIN SMALL LETTER E WITH ACUTE
...
        364   244   F4    ô      LATIN SMALL LETTER O WITH CIRCUMFLEX
...
```

首先，以严格的字节顺序排序文件：

```
$ LC_ALL=C sort french-english        以传统ASCII码顺序排序
cote    dimension
coté    dimensioned
côte    coast
côté    side
```

你应该会发现，正如其数值的情况：e（145_8）排在 é（351_8）之前；而 o（157_8）排在 ô（364_8）之前。

现在我们以 Canadian-French 的文本顺序排序：

```
$ LC_ALL=fr_CA.iso88591 sort french-english  以Canadian-French的locale排序
côte    coast
cote    dimension
coté    dimensioned
côté    side
```

输出的顺序完全不同于按照原始字节值所做的传统排序。

排序的惯例，完全视语言、国家以及文化而定，且这样的规则有时会非常复杂。即便是英文这种看起来与重音不相关的语言，都有复杂的排序规则。可以看看电话簿里，那些大小写、数字、空间、标点符号，还有姓名变化，例如 McKay 与 Mackay 的处理方式。

4.1.2 以字段排序

如果要进一步控制排序操作，可以用 `-k` 选项指定排序字段，并且用 `-t` 选项来选择字段定界符。

如未指定 `-t`，则表示字段以空白分隔且记录内开头与结尾的空白都将忽略；如指定 `-t` 选项，则被指定的字符会分隔字段，且空白是有意义的。因此一个包括“空白-X-空白”三个字符的记录，如果没有指定 `-t` 则只有一个字段，如果使用 `-t' '`，则为三个字段（第一个与第三个字段是空的）。

`-k` 选项的后面接着的是一个字段编号，或者是一对数字，有时在 `-k` 之后可用空白分隔。每个编号后面都可以接一个点号的字符位置，及/或修饰符（modifier）字母之一，如表 4-1 所示。

表 4-1：排序键值字段的类型

字母	说明
b	忽略开头的空白
d	字典顺序
f	不区分字母的大小写
g	以一般的符点数字进行比较，只适用于 GNU 版本
i	忽略无法打印的字符
n	以（整数）数字比较
r	倒置排序的顺序

字段以及字段里的字符是由 1 开始。

如果仅指定一个字段编号，则排序键值会自该字段的起始处开始，一直继续到记录的结尾（而非字段的结尾）。

如果给的是一对用逗点隔开的字段数字，则排序键值将由第一个字段值的起始处开始，结束于第二个字段值的结尾。

使用点号字符位置，则比较的开始（一对数字的第一个）或结束（一对数字的第二个）在该字符位置处：-k2.4,5.6指的是从第二个字段的第四个字符开始比较，一直比到第五个字段的第六个字符。

如果一个排序键值的起始正好落在记录的结尾处之后，则排序键值为空，且空的排序键值在排序时将优先于所有非空的键值。

当出现多个-k选项时，会先从第一个键值字段开始排序，找出匹配该键值的记录后，再进行第二个键值字段的排序，以此类推。

注意： -k选项在我们测试的所有系统上都可用，但sort也认得过时的旧式字段规格，在该定义上，字段与字符位置是从0开始编号。键值从字段*n*中的字符*m*开始，定义为：+*n*.*m*，以及键值以-*n*.*m*结束。举例来说，sort +2.1 -3.2，等同于sort -k3.2,4.3。如省略字符位置，则默认为0。因此，+4.0nr与+4nr表示相同的意义：一个数值型的键值，从第5个字段起始处开始，但反向（由大至小）排序。

我们可以在password范例文件上试试这些选项，以冒号隔开的第一个字段：用户名称，进行排序：

```
$ sort -t: -k1,1 /etc/passwd              以用户名称排序
bin:x:1:1:bin:/bin:/sbin/nologin
chico:x:12501:1000:Chico Marx:/home/chico:/bin/bash
daemon:x:2:2:daemon:/sbin:/sbin/nologin
groucho:x:12503:2000:Groucho Marx:/home/groucho:/bin/sh
gummo:x:12504:3000:Gummo Marx:/home/gummo:/usr/local/bin/ksh93
harpo:x:12502:1000:Harpo Marx:/home/harpo:/bin/ksh
root:x:0:0:root:/root:/bin/bash
zeppo:x:12505:1000:Zeppo Marx:/home/zeppo:/bin/zsh
```

如果要再进一步控制排序后的结果，可在字段选择器（field selector）内，加入一个修饰符字母，定义字段里的数据类型及排序顺序。这里显示按照反向顺序的UID来排序password文件的结果：

```
$ sort -t: -k3nr /etc/passwd            反向UID的排序
zeppo:x:12505:1000:Zeppo Marx:/home/zeppo:/bin/zsh
gummo:x:12504:3000:Gummo Marx:/home/gummo:/usr/local/bin/ksh93
groucho:x:12503:2000:Groucho Marx:/home/groucho:/bin/sh
```

```
harpo:x:12502:1000:Harpo Marx:/home/harpo:/bin/ksh
chico:x:12501:1000:Chico Marx:/home/chico:/bin/bash
daemon:x:2:2:daemon:/sbin:/sbin/nologin
bin:x:1:1:bin:/bin:/sbin/nologin
root:x:0:0:root:/root:/bin/bash
```

更精确的字段规格应为 -k3nr,3（也就是从字段3起始处开始，以数值类型反向排序，并结束于字段3的结尾），或是 -k3,3nr，甚至是 -k3,3 -n -r 也可以，由于 sort 会在遇到第一个非阿拉伯数字处停止收集数据，所以 -k3nr 也正确。

在我们的 password 范例文件里，有三个用户拥有共同的 GID（字段4），因此我们可以先以 GID 排序，再以 UID 排序，如下所示：

```
$ sort -t: -k4n -k3n /etc/passwd          以GID与UID排序
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
chico:x:12501:1000:Chico Marx:/home/chico:/bin/bash
harpo:x:12502:1000:Harpo Marx:/home/harpo:/bin/ksh
zeppo:x:12505:1000:Zeppo Marx:/home/zeppo:/bin/zsh
groucho:x:12503:2000:Groucho Marx:/home/groucho:/bin/sh
gummo:x:12504:3000:Gummo Marx:/home/gummo:/usr/local/bin/ksh93
```

-u 选项的好用是在于：它可以要求 sort 仅输出唯一的记录，而"唯一的"是指它们的排序键值字段匹配，即使在记录的其他地方有差异也无所谓。我们再利用password文件看一次，发现：

```
$ sort -t: -k4n -u /etc/passwd          以唯一的GID排序
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
chico:x:12501:1000:Chico Marx:/home/chico:/bin/bash
groucho:x:12503:2000:Groucho Marx:/home/groucho:/bin/sh
gummo:x:12504:3000:Gummo Marx:/home/gummo:/usr/local/bin/ksh93
```

注意，输出结果变短了：有三个用户都为组1000，但只有一个输出。我们会在4.2节中说明其他选出唯一记录的方式。

4.1.3 文本块排序

有时，你会需要将多行记录组合而成的数据排序。地址清单就是一个很好的例子，为了方便阅读，地址记录经常会切断，以一个或数个空行将彼此隔开。像这种数据，没有一定的排序键值位置可供 -k 选项使用，所以你得自救，提供一些额外标记（markup）给这些数据。这里是一个简单范例：

```
$ cat my-friends                          显示地址数据文件
# SORTKEY: Schloß, Hans Jürgen
```

```
Hans Jürgen Schloß
Unter den Linden 78
D-10117 Berlin
Germany

# SORTKEY: Jones, Adrian
Adrian Jones
371 Montgomery Park Road
Henley-on-Thames RG9 4AJ
UK

# SORTKEY: Brown, Kim
Kim Brown
1841 S Main Street
Westchester, NY 10502
USA
```

这里的排序小技巧，就是利用awk处理较一般性的记录分隔字符的能力，识别段落间隔，在每个地址内暂时使用一个未使用过的字符（例如使用一个无法打印的控制字符），取代分行，以及用换行字符取代段落间隔。sort看到的行就会变成这样：

```
# SORTKEY: Schloß, Hans Jürgen^ZHans Jürgen Schloß^ZUnter den Linden 78^Z...
# SORTKEY: Jones, Adrian^ZAdrian Jones^Z371 Montgomery Park Road^Z...
# SORTKEY: Brown, Kim^ZKim Brown^Z1841 S Main Street^Z...
```

在这里，^Z是一个Ctrl-Z字符。第一个过滤步骤是通过sort排序后恢复换行与段落的分隔符号，且排序键值行是容易删除的，如有需要，可使用grep轻松地删除它们。整个管道看起来就像这样：

```
cat my-friends |                                          在地址数据文件里的管道
  awk -v RS="" '{ gsub("\n", "^Z"); print }' |            转换地址为单个行
    sort -f |                                             排序地址数据，忽略大小写
      awk -v ORS="\n\n" '{ gsub("^Z", "\n"); print }' |   恢复行结构
        grep -v '# SORTKEY'                               删除标记行
```

函数gsub()功能为全局性替换（global substitution），类似sed下的s/x/y/g架构。RS变量是输入数据的记录分隔器（Record Separator）。通常输入数据是以换行字符隔开，使每一行成为单个的记录。“RS=""”是一个特殊用法，指的是记录以空行的方式隔开；例如每个块或文本段落自成一个记录。这就是我们的例子里输入的数据形式。最后，ORS指的是输出记录分隔器（Output Record Separator）；以print显示的每条输出记录会以其值作为终止。一般来说默认值为单个换行字符；在此设置它为"\n\n"，是为了保持用空白行分隔记录的输入格式（更多相关细节，请见第9章）。

这个管道应用在上述地址数据文件上的输出为：

```
Kim Brown
1841 S Main Street
```

```
Westchester, NY 10502
USA

Adrian Jones
371 Montgomery Park Road
Henley-on-Thames RG9 4AJ
UK

Hans Jürgen Schloß
Unter den Linden 78
D-10117 Berlin
Germany
```

这种方法最棒的地方就在于我们可以很轻松地把额外的键值放入到每条地址数据里，这么一来不论是在排序或选择时都可以使用，例如一个额外的标记行，形式如下：

```
# COUNTRY: UK
```

在每一个地址中，再将grep搭配管道操作：grep '# COUNTRY: UK' 置于sort前，就可以取出只有UK的地址数据做进一步处理了。

当然，你也可以更完善些，以更细节的XML标记作为地址数据各部分类型的识别：

```
<address>
  <personalname>Hans Jürgen</personalname>
  <familyname>Schloß</familyname><br/>
  <streetname>Unter den Linden<streetname>
  <streetnumber>78</streetnumber><br/>
  <postalcode>D-10117</postalcode>
  <city>Berlin</city><br/>
  <country>Germany</country>
</address>
```

有了这种较为华丽的数据处理过滤程序，你可以先以国家与邮政编码排序好邮件，让邮局在处理时更为顺畅，不过我们前面提到的小标记与简单管道处理方式，多半就足以完成此工作了。

4.1.4 sort的效率

排序数据的操作，很明显地就是比较所有成对的项目，看哪个在先哪个在后，因此得到为人所熟知的算法，如冒泡排序（bubble sort）与插入排序（insertion sort）。这些讲求快速排序（quick-and-dirty）的算法，在处理少量数据时很好用，不过当面临大量的数据需要处理时速度就不够快了，因为如果需要排序n条记录，它们会让数据增加到几乎是n^2（平方）那么大。这和我们在本书里所讨论的大部分过滤器完全不同：后者是读取一条记录，处理它并输出它，所以它们的执行时间完全与记录的数量n成比例。

幸运的是，计算领域中的很多人在关心排序的问题，有一些不错的排序算法，例如从复杂度来看，有 $n^{3/2}$ 次方（Shellsort）、$n \log n$（heapsort、mergesort 及 quicksort）。从针对受限制的数据种类来看，有 n 的分布排序（distribution sort）。UNIX 的 sort 命令实现，已有许多人在进行研究并优化调整：你可以相信它的工作效率，它一定可以比你自己做得好，而且几乎不必学习一堆的排序算法。

4.1.5 sort 的稳定性

在排序算法里有个重要的问题：是否稳定（stable）？这个问题指的是：相同的记录输入顺序是否在输出时也可保持原状？当你以多键值为记录进行排序，或是以管道处理时，排序稳定性就非常重要了。POSIX 不需要这个所谓的 sort 的稳定性，绝大多数的实现也不需要，来看看下面这个范例：

```
$ sort -t_ -k1,1 -k2,2 << EOF              以前面两个字段为键值，排序这四行
> one_two
> one_two_three
> one_two_four
> one_two_five
> EOF
one_two
one_two_five
one_two_four
one_two_three
```

每条记录内的排序字段都相同，但输出却与输入不一致，所以我们说 sort 并不稳定。幸好：GNU 实现了 coreutils 包（注 1）弥补不足，它可以通过 --stable 选项补救此问题：设置此选项，上例的输出便可与输入一致了。

4.1.6 sort 小结

sort 绝对排得进 UNIX 重要命令前十名：把它学好一定没错，因为你会经常用到。本章一开始，就详细介绍过 sort 了，不过你可以参考计算机里的 *sort*(1)手册页（manual page），来了解更多用法。sort 当然经过 POSIX 的标准化，所以几乎在所有机器上都能使用。

4.2 删除重复

有时，将数据流里连续重复的记录删除是有必要的。我们在 4.1.2 节里介绍过 sort -u 的用法，不过我们也知道，它的消除操作是依据匹配的键值，而非匹配的记录。uniq

注 1：　可到 *ftp://ftp.gnu.org/gnu/coreutils/* 下载。

命令提供另一种过滤数据的方式：它常用于管道中，用来删除已使用sort排序完成的重复记录：

```
sort ... | uniq | ...
```

uniq有3个好用选项：-c可在每个输出行之前加上该行重复的次数，这部分我们在第5章例5-5的单词出现频率过滤器里会用到；-d选项则用于仅显示重复的行；而-u仅显示未重复的行。下面是一些范例说明：

```
$ cat latin-numbers                     显示测试文件
tres
unus
duo
tres
duo
tres

$ sort latin-numbers | uniq             显示唯一的、排序后的记录，重复则仅取唯一行
duo
tres
unus

$ sort latin-numbers | uniq -c          计数唯一的、排序后的记录
      2 duo
      3 tres
      1 unus

$ sort latin-numbers | uniq -d          仅显示重复的记录
duo
tres

$ sort latin-numbers | uniq -u          仅显示未重复的记录
unus
```

uniq有时会拿来与diff工具搭配应用，在找出两个相似数据流的异同的时候很方便，例如字典单词列表、在映射目录树内的路径、电话簿等等。在大部分的实现中有很多其他的选项可供使用，你可以在uniq(1)手册页里找到相关描述，不过它们很少使用。而uniq就像sort一样，已被POSIX标准化，所以在哪儿都能使用。

4.3 重新格式化段落

大部分功能强大的文本编辑器都提供重新格式化段落的命令，供用户切分段落，使文本行数不要超出我们看得到的屏幕范围；我们写这本书时就用了许多类似命令。有时你在Shell脚本内处理数据流时需要完成重新格式化，或者在一个缺乏重新格式化命令但提供了Shell转义的编辑器内完成它。在不提供此功能的情况下，你可以使用fmt命令。虽然POSIX未提及fmt，不过你还是可以在许多现行的UNIX系统下找到它；如果你的旧系统里没有fmt，只要安装GNU的coreutils包即可。

虽然一些 fmt 的实现有较多的选项可用，但其实我们发现只有两种较常用到：-s 仅切割较长的行，但不会将短行结合成较长的行，而 -w *n* 则设置输出行宽度为 *n* 个字符（默认通常约 75 个左右）。这里的几个例子是一字一行的拼音字典：

```
$ sed -n -e 9991,10010p /usr/dict/words | fmt      重新格式化20个字典单词
Graff graft graham grail grain grainy grammar grammarian grammatic
granary grand grandchild grandchildren granddaughter grandeur grandfather
grandiloquent grandiose grandma grandmother

$ sed -n -e 9995,10004p /usr/dict/words | fmt -w 30   重新将10个单词格式化为短的行
grain grainy grammar
grammarian grammatic
granary grand grandchild
grandchildren granddaughter
```

如果你的系统里没有 /usr/dict/wordsr，那么有可能文件是在 /usr/share/dict/words 或 /usr/share/lib/dict/words。

仅作切割的选项：-s，在你想将长的行绕回，短的行保持不动时很好用，这么做也能使结果与原始版本间的差异达到最小：

```
$ fmt -s -w 10 << END_OF_DATA                仅重新格式化长的行
> one two three four five
> six
> seven
> eight
> END_OF_DATA
one two
three
four five
six
seven
eight
```

警告： 你可能觉得，使用 fmt -w 0 就能将输入数据流切为一字一行；或者你想用较长的宽度，将所有的字符串在一起，并删除分行。遗憾的是，fmt 的每个版本，都有不同的行为：

- 旧版的 fmt 不提供 -w 选项，它们使用 -n 指定宽度为 n 个字符宽。
- 所有版本都禁止将宽度设为零，不过接受 -w 1 或 -1 的用法。
- 所有版本都保留开头的空白。
- 有些版本会保留类似邮件标题的行。
- 有些版本会保留开头为一个点号的行（troff 排字命令）。
- 宽度限制各有不同。我们发现的就有好几种：Solaris 的上限是 1021、HP/UX 11 的是 2048、AIX 与 IRIX 的是 4093、OSF/1 4.0 的是 8189、OSF/1 5.1 的是 12285，还有 FreeBSD、GNU/Linux 与 Mac OS（最大的 32 位带符号的整数）的是 2147483647。

- NetBSD 与 OpenBSD 的 fmt 版本有不同的命令行语法，看起来应是分配一个缓冲区以保留输出行，因为它们会针对大的宽度值给出 out of memory 这样的信息。
- IRIX 的 fmt 位于 /usr/sbin，此目录不太可能出现在你的查找路径里。
- HP/UX 在 11.0 之前的版本没有 fmt。

由于这些fmt版本各有不同的特性，使得它很难用于对可移植性要求很高的脚本中或是复杂的重新格式化工作中。

4.4 计算行数、字数以及字符数

我们已使用过字数计算工具 wc 好几次了。它可能是 UNIX 工具集里最古老也最简单的工具程序，同时 POSIX 也已将它标准化。wc 的默认输出是一行报告，包括行数、字数以及字节数：

```
$ echo This is a test of the emergency broadcast system | wc      计数报告
      1       9      49
```

要求仅输出部分结果时，可使用的选项有：-c（字节数）、-l（行数）以及 -w（字数）：

```
$ echo Testing one two three | wc -c      计算字节数
22

$ echo Testing one two three | wc -l      计算行数
1

$ echo Testing one two three | wc -w      计算字数
4
```

-c 选项原本是表示字符数（character count），但因为有多字节字符集的编码存在 —— 像是 UTF-8，因此在当前系统上，字节数已不再等同于字符数了，也因此，POSIX 出现了 -m 选项，用以计算多字节字符，对 8 位字符数据而言，它是等同于 -c 的。

虽然 wc 最常处理的是来自于管道的输入数据，但它也接受命令行的文件参数，可以生成一行一个结果，再附上报告：

```
$ wc /etc/passwd /etc/group              计算两个文件里的数据
    26      68   1631 /etc/passwd
 10376   10376 160082 /etc/group
 10402   10444 161713 total
```

wc 的现代版本会随 locale 而有不同结果：将环境变量 LC_CTYPE 设为想用的 locale，会影响 wc 把字节序列解释为字符或单词（word）分隔器。

第 5 章我们会说明另一个用来报告每个单词发生频率的相关工具：wf。

4.5 打印

和计算机比起来，打印机是比较慢的设备，因为它们一般都是共享的，通常不希望用户直接传递工作给它们；因此，大部分的操作系统都提供命令，让用户传送需求到打印*daemon*中（注2），daemon将打印的工作放进队列，并处理打印机与队列管理。打印命令的处理可以很快，因为打印是在所需的资源呈现可用状态时，才能在后台中被执行。

UNIX里支持的打印功能包括了两类不同的命令，但拥有相同的功能，见表4-2。商用UNIX系统与GNU/Linux通常两种都支持，不过BSD系统就仅支持Berkeley风格。POSIX则只定义了`lp`命令。

表4-2：打印的命令

Berkeley	System V	用途
lpr	lp	传送文件到打印队列
lprm	cancel	从打印队列中删除文件
lpq	lpstat	报告队列状态

以下为上述命令的例子，首先是Berkeley风格：

```
$ lpr -Plcb102 sample.ps                    将PostScript文件传给打印队列lcb102

$ lpq -Plcb102                              要求查看打印队列状态
lcb102 is ready and printing
Rank    Owner   Job     File(s)     Total Size
active  jones   81352   sample.ps   122888346 bytes

$ lprm -Plcb102 81352                       停止此进程！结束这个庞大的作业
```

然后是System V风格，如下：

```
$ lp -d lcb102 sample.ps                    传送PostScript文件到打印队列lcb102
request id is lcb102-81355 (1 file(s))

$ lpstat -t lcb102                          要求查看打印队列状态
printer lcb102 now printing lcb102-81355

$ cancel lcb102-81355                       噢！不要打印该工作 ！
```

`lp`与`lpr`当然也可以用来读取来自标准输入的数据，而不是来自命令行上的文件，所以它们也常用在管道的结尾。

系统管理可以将特定单个队列设为系统默认值，所以当默认值是可接受时，无须提供队

注2： daemon（读作dee-mon）是一个长期处于执行状态的进程，提供诸如账号管理、文件访问、登录、网络连接、打印与时刻等服务。

列名称。每个用户都能设置环境变量：PRINTER（Berkeley）或LPDEST（System V），选择个人的默认打印机。

打印队列名称是由各站点指定（site-specific）的：小站点可直接称为printer，并将其设为默认值。大站点则可挑选反映其位置的名称，例如建筑物缩写以及房间编号；或者是识别特定打印机类型或功能，像是bw指的是黑白打印机，而color则为比较昂贵的打印机。

麻烦的是，使用现代网络化的智能型打印机的时候，lprm、cancel、lpq以及lpstat这样的命令已经不像以前那么有用了：打印工作很快就传到打印机，并出现在打印机daemon上，显示打印好了，然后从打印队列中删除——即使打印机仍然将它们搁在内存或是文件系统里，这时，它仍能同时处理其他的打印工作。就这点来看，唯一要用到的资源，就是利用打印机的控制面板，取消不想要的工作。

4.5.1 打印技术的演化

从UNIX开发以来，打印机的技术已有长足的进展与改变。这个产业一开始是金属字符捶打色带与纸的大型图文件打印机与电子打字机，后来是电子式、点矩阵、喷墨以及激光打印机，打印字符越来越细致。

微处理器的进步，允许在打印机内直接实现简易命令语言，例如Hewlett-Packard Printer Command Language（PCL）以及HP Graphics Language（HPGL），当然也有功能更齐全的程序语言，最有名的就是Adobe PostScript了。Adobe Portable Document Format（PDF）是PostScript的后续版本，它更简洁，但不能编程。PDF还提供额外的功能，例如彩色幻灯片、数码签名、文件访问控制、加密、高级数据加密以及独立页面（page independence）。近期出现的新功能是，使用高性能打印机将多个页面同时印成一页，以及可使用PDF浏览器，迅速地浏览所需页面。

最新一代的设备，将打印、影印及扫描整合到一个系统上，并结合了磁盘文件系统以及网络访问，支持多页面描述语言与图形文件格式，甚至还出现了以GNU/Linux作为嵌入式操作系统的设备。

遗憾的是，UNIX打印软件的改进速度并没有这些打印技术改良得快，而且在利用命令层级访问较新打印机功能上还是很缺乏。有两个著名的软件项目试图解决这样的窘境：它们是Common UNIX Printing System（CUPS，注3）和lpr next generation（LPRng，注4）。许多大型UNIX站点在这两者之中取其一；这两种软件都提供熟悉的UNIX打印命令，但带有更多的选项。这两种软件也都充分支持PostScript与PDF文件的打印：必

注3：　见*http://www.cups.org/*及本书参考书目中所列的书籍。

注4：　见*http://www.lprng.org/*。

要时，它们会利用Aladdin或GNU ghostscript解释器，帮助功能不足的打印机把这类文件转换为其他的格式。CUPS也支持各类图形图像文件格式的打印，以及一次*n*页（将几个图形缩小，打印在同一张纸上）打印的功能。

4.5.2 其他打印软件

不要被pr的名字欺骗了，它其实并不是打印文件用的命令，它不过是过滤数据为打印做准备。以最简单的情形来说，pr会以文件的修改时间作为页面标题的时间戳；如果输入是自管道而来，则使用当前的时间，接上文件名称（如果输入的数据内容在管道中，则为空的）以及页码，以每页固定行数（66）的方式打印。也就是这样：

```
pr file(s) | lp
```

会显示适当的列表。不过，自从20世纪70年代古老的机械式打印机退役之后，这种简化的方式就不再有效了。每种打印机默认的字体大小与行列空间都不同，而且平常使用的纸张大小也都不一样。

pr

语法

```
pr [ options ] [file(s) ]
```

用途

将文本文件编页，供打印用。

主要选项

-c*n*

产生*n*栏（column）的输出。此选项可以缩写为 -*n* 替代（例：-4意同于 -c4）。

-f

在首页之后的每一页标题前置一个ASCII分页字符（formfeed character）标题。此选项在FreeBSD、NetBSD与Mac OS X里为 -F；在OpenBSD里则两种都可以。POSIX里一样两种都能用，只是意义稍有不同。

-h *althdr*

将页标题（page header）内的文件名称，改用字符串 *althdr* 取代。

-l*n*

产生*n*行的页面。有些版本将页首行与页尾行计算在内，有些则不是。

-o*n*

输出位移*n*个空白。

pr（续）

-t

不显示标题。

-w*n*

每行至多*n*个字符。以单栏输出而言，如有需要会将较长的行切分绕回至另外一行；否则，在多栏输出的情况下，会截去长的行以符合指定。

行为模式

pr会读取指定的文件，如果未给予指定文件，则读取标准输入，再将编页完成的数据写到标准输出。

警告

pr的各个版本，对支持选项与输出格式有极大的差异；使用GNU coreutils版本可让用户在所有系统上使用时，都能得到一致的行为模式。

反而比较常用的是：-l选项设置输出页面长度、-w设置页面宽度，-o设置文本位移。另外还有-f也是必备的（有些系统是-F）——用来在首页后的每页页标题加入ASCII分页控制字符，这是为了保障每页页标题都起始于新的一页。所以实际上你应该会这样用：

```
pr -f -l60 -o10 -w65 file(s) | lp
```

如果你稍后使用不同的打印机，必须更改这些数值型参数，这点让pr很难应用在必须具备可移植性的Shell脚本上。

pr有一个功能可通用于多数情况下：以-c*n*选项要求*n*栏输出。如果搭配-t选项可省略页标题，这样就可以产生适当的多栏列表。下面这个例子便是将26个单词格式化为5栏的状态：

```
$ sed -n -e 19000,19025p /usr/dict/words | pr -c5 -t
reproach       repugnant      request        reredos        resemblant
reptile        repulsion      require        rerouted       resemble
reptilian      repulsive      requisite      rerouting      resent
republic       reputation     requisition    rescind        resentful
republican     repute         requited       rescue         reserpine
repudiate
```

如果栏宽太小，pr会默默截去超出的数据，以避免该行过长。我们可以试试看将上例26个单词格式化为10栏（截断），如下所示：

```
$ sed -n -e 19000,19025p /usr/dict/words | pr -c10 -t
reproa republ repugn reputa requir requit rerout rescue resemb resent
```

```
reptil republ repuls repute requis reredo rescin resemb resent reserp
reptil repudi repuls reques requis rerout
```

pr 有很多选项可用，长久以来，在各种 UNIX 系统上，它们的选项的使用、输出格式与每页可打印行数上有着各种不同之处。我们建议使用 GNU coreutils 包的版本，因为这么一来到哪儿都能有一致的界面，可以用的选项也比其他版本为多。你可以参考 pr(1) 的手册页了解更多的细节。

虽然有些 PostScript 的打印机接受纯文本，但大部分仍是不接受的。这时像 TEX 与 troff 这类的排版系统，便能将标记文件转为 PostScript 或是 PDF（或两者都可）的页面映像。如果你手上只有纯文本文件，要怎么打印呢？UNIX 的打印系统会调用适当的过滤程序执行转换，不过这么一来，你就不能控制它显示出来的样子了。这类问题的解决方法就是文本到 PostScript 的过滤器，例如 a2ps（注 5）、lptops（注 6）或 Sun Solaris 专属的 mp。用法如下：

```
a2ps file > file.ps                    产生文件的 PostScript 列表
a2ps file | lp                         打印文件的 PostScript 列表

lptops file > file.ps                  产生文件的 PostScript 列表
lptops file | lp                       打印文件的 PostScript 列表

mp file > file.ps                      产生文件的 PostScript 列表
mp file | lp                           打印文件的 PostScript 列表
```

这三种方式都提供命令行选项，以选择字体、指定字体大小、提供或取消页首，以及选择多栏输出功能。

BSD、IBM AIX 及 Sun Solaris 系统还提供 vgrind 命令（注 7），它用来过滤以各种程序语言构成的文件，将它们转换为 troff 输入：斜体字为注释、粗体为关键字，并且将目前的功能注释在边缘；将这样的数据进行排版，输出为 PostScript。随之衍生而来的 tgrind（注 8）也做类似的工作，只不过有更多的字体选项、行编号、索引，并且支持更多的程序语言。tgrind 产生的是 TEX 输入，可迅速产生 PostScript 与 PDF 输出。图 4-1 即为其简单输出范例。这两种程序都可以轻松应用于排版程序列表的打印。

```
$ tgrind -p hello.c                    排版与打印 hello.c

$ tgrind -i 1 -fn Bookman -p hello.c   打印图 4-1 所显示的列表

$ vgrind hello.c | lp                  排版与打印 hello.c
```

注 5：见 *ftp://ftp.gnu.org/gnu/a2ps/*。

注 6：见 *http://www.math.utah.edu/ub/lptops/*。

注 7：见 *http://www.math.utah.edu/pub/vgrind/*。

注 8：见 *http://www.math.utah.edu/pub/tgrind/*。

(hello.c)

```
#include <stdio.h>
#include <stdlib.h>

const char *hello(void);
const char *world(void);

int
main(void)                                                        main
{
    (void)printf("%s, %s\n", hello(), world());
    return (EXIT_SUCCESS);    /* use ISO Standard C exit code */
}

const char *
hello(void)                                                       hello
{
    return ("hello");
}

const char *
world(void)                                                       world
{
    return ("world");
}
```

Linenumber Index

hello 15	main 8	world 21

19:18 Apr 19 2004 *Page 1 of hello.c*

图 4-1：用 tgrind 排版的一个著名 C 语言程序

4.6 提取开头或结尾数行

有时，你会需要从文本文件里把几行字 —— 多半是靠近开头或结尾的几行，提取出来。例如本书 XML 文件的章节标题，就全出现在每个文件的前几行；或者，有时你只要瞧瞧工作日志的后面几行，就可以了解最近工作活动的大概情况。

这两种操作都很简单，你可以用下面这几招，显示标准输入前 *n* 条记录，或是命令行文件列表中的每一个的前 *n* 条记录：

```
head -n n      [ file(s) ]
head -n        [ file(s) ]
awk 'FNR <= n' [ file(s) ]
sed -e nq      [ file(s) ]
sed nq         [ file(s) ]
```

POSIX 要求 head 选项需设为 -n 3，不接受 -3，但我们测试过的每个版本，这两种设置方式都可以。

如果只有单个编辑命令时，sed 允许省略 -e 选项。

如果这里显示的行少于 *n*，其并非有误。

要显示结尾数行，可以这么做：

```
tail -n n      [ file ]
tail -n        [ file ]
```

就像 head 一样，POSIX 只指定第一种形式，但其实两种用法在我们使用的所有系统里都是可接受的。

说也奇怪，head 可以处理命令行上的数个文件，但 POSIX 及传统的 tail 却不行，幸好这个不便之处在现代所有的 tail 版本都已修正。

在交互式 Shell 通信期中，有时需要监控某个文件的输出 —— 如日志这类持续写入状态的文件。-f 选项这时就派上用场了，它可以要求 tail 显示指定的文件结尾行数，接着进入无止尽的循环中 —— 休息一秒后又再度醒来并检查是否需显示更多的输出结果。在设置 -f 的状态下，tail 只有当你中断它时才会停止 —— 通常是输入 Ctrl-C 来中断：

```
$ tail -n 25 -f /var/log/messages          观察不断成长的系统信息日志
...
^C                                         Ctrl-C 停止 tail
```

由于 tail 加上 -f 选项后便不会自己中断，所以此选项不可用于 Shell 脚本。

因为 tail 的工作必须一直维护最近记录的历史，因此使用 awk 或 sed 时，没有更短或更简单的方式可以替代 tail。

虽然我们无法在这里详细解释这些工具，不过你应留意下面的几个命令，这些命令在本书里的一些小范例都有用到，值得你加入工具箱。

- dd以用户指定的块大小与数量拷贝数据。不过它也具有一些有限的能力，可进行大小写转换，以及ASCII与EBCDIC间的转换。以字符集转换而言，现代的、POSIX标准的iconv命令可以将文件从一种编码集（code set）转换成另一种更具灵活性的字码集。
- file将其参数文件内容的前几个字节，与样式数据库进行比对，再在标准输出下，针对各文件显示一行简短报告。绝大多数厂商提供的file版本都可以识别100种左右的文件类型，不过无法分辨来自其他UNIX风格的二进制可执行文件与对象文件，或是来自其他操作系统的文件。还有一种更好用的开放源代码版本（注9），由于有许多人的贡献，让我们可以享受它的便利：它识别的文件类型有1200种之多，包含许多非UNIX操作系统下的文件。
- od为八进制码转储（octal dump）命令，显示ASCII码、八进制以及十六进制的字节数据流。可以在命令行选项中设置要读取的字节数，也可选择输出格式。
- strings针对输入数据查找以换行符号或NUL结尾的四个（或以上）可打印字符的序列，再将结果打印至标准输出。多半用来查看二进制文件——例如编译后的程序或是数据文件的内部信息。桌面应用软件、图像以及声音文件有时会在文件的开头处包含一些有用的文本数据，而GNU的head还提供一个方便的-c选项，让用户用以限制输出的字符数：

```
$ strings -a horne01.jpg | head -c 256 | fmt -w 65      查看天文学图像文件
JFIF Photoshop 3.0 8BIM Comet Hale-Bopp shows delicate
filaments in it's blue ion tail in this exposure made Monday
morning 3/17/97 using  12.5 inch F/4 Newtonian reflecting
telescope. The 15 minute exposure was made on Fujicolor SG-800
Plus film. 8BIM 8BI
```

4.7 小结

本章总共介绍了约30种处理文本文件的好用工具。它们都是功能很强的工具组，可用来编写Shell脚本。最重要也最复杂的就是sort了。而fmt、uniq，以及wc这些命令则常出现在管道里，用来简化或摘要数据。当你想迅速浏览一堆不熟悉的文件时，file、head、strings以及tail都会是你的好帮手。最后，a2ps、tgrind以及vgrind可以用来将程序内容列表输出——当然也包含Shell脚本，从而让你更易于阅读。

注9： 可从*ftp://ftp.astron.com/pub/file/*获取。

第5章

管道的神奇魔力

本章我们要解决的是一些相当简单的文字处理工作。在本章所有例子里，最有趣的就是这些脚本都是由简单的管道构建而成：命令一个个串联起来完成任务，而每一个又都完成步骤性的一个重要任务。

当你在UNIX里对付文字处理作业时，必须谨记一个UNIX工具使用原则就是：想清楚这个问题该如何划分为更简单的工作，每个部分是不是已有现成的工具能解决，还是你可以写几行Shell程序或使用脚本语言就能马上解决。

5.1 从结构化文本文件中提取数据

在UNIX下的管理性文件，大部分都是无须使用任何特殊的文件专用工具，即可编辑、打印与阅读的简易文本文件。这些文件大部分放在标准目录：/etc下。最常见的例子就是密码文件与组文件（passwd与group）、文件系统加载表（fstab或vfstab）、主机文件（hosts），以及默认的Shell启动文件（*profile*），以及系统启动与关机的Shell脚本（存放在子目录树rc0.d、rc1.d…rc6.d之下，也有可能是其他目录）。

文件格式通常在UNIX使用手册的第5节（Section 5）说明，所以执行man 5 passwd就能看到/etc/passwd相关的结构信息（注1）。

尽管它叫做密码文件，但它仍开放给所有人读取。也许应该叫它用户文件，因为它的内容是有关系统里各用户的基本信息的，将这些信息集结在一起，一行表示一个账号，再以冒号隔开各信息字段。文件的格式在3.3.1节里已经提过，下面我们来看几个典型的条目（entry）：

注1： 有些系统的文件格式是在第7节（Section 7），因此在这种情况下应是执行man 7 passwd。

```
jones:*:32713:899:Adrian W. Jones/OSD211/555-0123:/home/jones:/bin/ksh
dorothy:*:123:30:Dorothy Gale/KNS321/555-0044:/home/dorothy:/bin/bash
toto:*:1027:18:Toto Gale/KNS322/555-0045:/home/toto:/bin/tcsh
ben:*:301:10:Ben Franklin/OSD212/555-0022:/home/ben:/bin/bash
jhancock:*:1457:57:John Hancock/SIG435/555-0099:/home/jhancock:/bin/bash
betsy:*:110:20:Betsy Ross/BMD17/555-0033:/home/betsy:/bin/ksh
tj:*:60:33:Thomas Jefferson/BMD19/555-0095:/home/tj:/bin/bash
george:*:692:42:George Washington/BST999/555-0001:/home/george:/bin/tcsh
```

再来复习一下，密码文件的 7 个字段分别是：

1. 用户名称
2. 加密的密码，或指出密码存储于另外一个文件中
3. 用户 ID（user ID）数字
4. 用户组 ID（group ID）数字
5. 用户姓名，或其他相关数据（办公室号码、电话等）
6. 根目录
7. 登录的 Shell

每个字段几乎对各种不同的 UNIX 程序都很重要，第 5 个字段除外。传统上，第 5 个字段用来置放用户相关信息。其实原本它叫做 gecos 字段，这个名称有历史原因，它是在 20 世纪 70 年代在贝尔实验室时加入的，当初是为了让 UNIX 系统能与其他运行通用电子综合操作系统（General Electric Comprehensive Operating System）的计算机进行通信而产生的，后者需要 UNIX 用户相关的额外信息。今天，多数站点将它用来存放用户姓名，所以我们把它简称为姓名字段。

以这个范例而言，我们假定本地端站点在姓名字段里存有其他信息：建筑物与办公室号码（例如范例第一条记录里的 OSD211），以及电话号码（555-0123），且这些数据与个人姓名以斜杠分隔。

我们可以利用像这样的文件，编写一些软件，建立一份办公室名录。通过这种方式，只需要维持一份文件 /etc/passwd的最新状态，当主文件改变时，衍生文件便会产生，更机动的方法是使用 cron，让它隔一段时间执行一次（我们将在 13.6.4 节中讨论 cron）。

我们的首度尝试，是要做出一个简单的办公室名录文本文件，完成后应该是这样：

```
Franklin, Ben                        •OSD212•555-0022
Gale, Dorothy                        •KNS321•555-0044
...
```

其中，• 表示 ASCII 的制表字符（Tab）。在姓名上，我们沿用传统名录顺序（姓在先），

用空白填补姓名字段，让每个姓名字段都有固定长度。然后，在办公室编号与电话号码前放置制表字符（Tab），以保留某种便利的结构，让其他工具也能善加利用此数据。

像awk这样的脚本语言的设计就是为了让这类工作更简单，因为它们提供自动化的输入处理，并将输入记录分割成字段，所以我们可以完全在这样的语言中编写转换工作。不过，我们希望展现的是：如何使用其他 UNIX 工具达到相同目的。

就密码文件的各行来看，我们需要提取字段 5，将它分割为三个子字段，再重新安排将姓名放在第一个子字段，接着写入一个办公室名录行，供排序进程使用。

awk 与 cut 是提取字段的好工具：

```
... | awk -F: '{ print $5 }' | ...
... | cut -d: -f5 | ...
```

稍稍复杂的地方在于我们有两个字段要处理，为简单化，我们希望它们保持分隔，但又需要结合它们的输出，以便产生名录记录。join 命令符合我们的需求：它期待两个输入文件，且这两个文件里的记录都以相同的唯一键值排序，将共享相同键值的行结合后，产生单一输出行，并由用户控制要输出的字段。

由于我们的名录要包含三个字段，因此使用 join 时要建立三个中间文件，这三个中间文件包含以冒号隔开的 *key:person*、*key:office* 以及 *key:telephone*，每一对对应一行。这些可以是临时性文件，因为它们可以自动从密码文件中衍生出来。

那么，我们要用的键值（key）是什么?它只要是唯一的即可。所以原始密码文件的记录编号也可以，但在这里，我们可以将用户名称（username）作为键值，因为我们知道密码文件里的用户名称必须是唯一值，且对我们而言，这个值比数字有意义多了。之后，如果我们想在名录里加上额外信息，例如工作职称，即可使用 *key:jobtitle* 建立另一个非临时性的文件，然后把它加入处理程序。

这种将程序视为过滤器、读取标准输入、写到标准输出的做法，与将输入、输出文件名直接编码到程序（hardcoding）的做法相比较，更具灵活性。对于较不常用的命令，最好是在程序里写上用途说明，而不是用简单或者隐密性的名称简略交代过去。因此，我们的 Shell 程序一开始就会像这样：

```
#! /bin/sh
# 过滤 /etc/passwd 这类格式的输入流，
# 并从此数据衍生出办公室名录。
#
# 语法：
#       passwd-to-directory < /etc/passwd > office-directory-file
#       ypcat passwd | passwd-to-directory > office-directory-file
#       niscat passwd.org_dir | passwd-to-directory > office-directory-file
```

由于密码文件是所有人都可读取的，任何由此文件导出的数据也是这样，因此事实上我们并不需要限制这个程序中间文件的访问权限。不过，由于我们绝大多数处理的都是敏感性数据，所以在开放程序时应养成一个好习惯：仅允许需要用到这个文件的用户或进程访问它。我们因此将把 umask（见附录B）重新设置为程序中的第一个操作：

```
umask 077                                    限制临时性文件只有我们可以访问
```

为了解文件任务且便于调试，让临时性文件具有部分共通名称会比较方便，这样一来，也可以避免因为这些文件而弄乱当前目录：我们在命名时将它们的文件名前都放置 /tmp/pd.。此外，为避免程序的多个实例在同一时间执行时发生名称冲突，我们也必须使其名称具有唯一性，在这里也就是使用进程编号：可以在 Shell 变量 $$ 里使用，将其放置在结尾，可通过字尾区别它们（$$ 的用法详见第10章）。因此，我们定义这些 Shell 变量，表示我们的临时性文件：

```
PERSON=/tmp/pd.key.person.$$                 具唯一性的临时性文件名
OFFICE=/tmp/pd.key.office.$$
TELEPHONE=/tmp/pd.key.telephone.$$
USER=/tmp/pd.key.user.$$
```

当工作终止时，无论是正常或异常终止，我们都要让临时性文件消失，因此使用 trap 命令：

```
trap "exit 1"                                HUP INT PIPE QUIT TERM
trap "rm -f $PERSON $OFFICE $TELEPHONE $USER" EXIT
```

在开发步骤，我们仅将第二个 trap 命令加上注释，以保留临时性文件供稍后检查（trap 命令在13.3.2节里会详细介绍。在此，了解当脚本离开时，trap 命令会使用给定的参数以自动执行 rm 就够了）。

我们需要重复执行取出字段1与5的操作，一旦得到这些信息，就无须再从标准输入取得输入流了，所以我们可以先将它们取出后放进临时性文件：

```
awk -F: '{ print $1 ":" $5 }' > $USER            读取标准输入
```

我们先作 *key:person* 这组文件，配合两步骤的 sed 程序，再接上一行简单的 sort；sort 命令，在4.1节里已介绍过。

```
sed -e 's=/.*==' \
    -e 's=^\([^:]*\):\(.*\) \([^ ]*\)=\1:\3, \2=' <$USER | sort >$PERSON
```

这段脚本使用 = 作为 sed 命令的分隔字符，因为斜杠和冒号在数据内容里都有了。第一个编辑操作是将第一个斜杠直至行结尾的所有数据提取出来，例如，如下的一行：

```
jones:Adrian W. Jones/OSD211/555-0123          输入行
```

处理后，成为：

```
jones:Adrian W. Jones                    首次编辑后的结果
```

第二个编辑就稍复杂些：匹配记录里的三个子模式。第一段：`^\([^:]*\)`匹配用户名称字段（例：`jones`）；第二段：`\(.*\)□`匹配文字到空白处（例：`Adrian□W.□`；□表示一个空格字符）；最后一段`\([^□]*\)`匹配记录里剩下的非空白文字（例如：`Jones`）。将置换后的文字重新整理出匹配的数据，产生像`Jones,□Adrian W.`这样的结果，这行`sed`命令所产生的结果就是我们预期的新顺序：

```
jones:Jones, Adrian W.                   显示第二个编辑的结果
```

下一步要作的是 *key:office* 文件：

```
sed -e 's=^\([^:]*\):[^/]*/\([^/]*\)/.*$=\1:\2=' < $USER | sort > $OFFICE
```

结果是列出用户与办公室信息：

```
jones:OSD211
```

再来的 *key:telephone* 操作也差不多：只要将匹配模式稍做调整即可：

```
sed -e 's=^\([^:]*\):[^/]*/[^/]*/\([^/]*\)=\1:\2=' < $USER | sort > $TELEPHONE
```

在这个步骤，已有了三个单个文件，都已完成排序，也都含有键值（也就是用户名称）、冒号，以及特定数据（姓名、办公室编号与电话号码）。`$PERSON`文件的内容看起来会像这样：

```
ben:Franklin, Ben
betsy:Ross, Betsy
...
```

`$OFFICE`文件里包含用户名称与办公室数据：

```
ben:OSD212
betsy:BMD17
...
```

`$TELEPHONE`文件记录了用户名称与电话号码：

```
ben:555-0022
betsy:555-0033
...
```

`join`的默认行为是输出共同键值，然后是第一个文件里这行剩余的字段，紧接着来自第二个文件里剩余的字段。这个共同键值默认为第一个字段，不过这可以通过命令行选项来修改：我们这里不需要此功能。通常`join`是以空格（space）来分隔字段，不过我们可以使用`-t`选项更改分隔符：在这里我们使用`-t:`。

join 使用一个五步骤的管道完成，过程如下：

1. 结合个人信息与办公室位置：

   ```
   join -t: $PERSON $OFFICE | ...
   ```

   这个运算的结果将成为下一步骤的输入，如下所示：

   ```
   ben:Franklin, Ben:OSD212
   betsy:Ross, Betsy:BMD17
   ...
   ```

2. 加入电话号码：

   ```
   ... | join -t: - $TELEPHONE | ...
   ```

   这里的操作结果一样也会成为下一步骤的输入，如下所示：

   ```
   ben:Franklin, Ben:OSD212:555-0022
   betsy:Ross, Betsy:BMD17:555-0033
   ...
   ```

3. 删除键值（也就是第一个字段），因为我们不再需要它。最简单的方式就是使用 cut，而这里的范围是指“使用字段 2 直到最后”，如下所示：

   ```
   ... | cut -d: -f 2- | ...
   ```

   这个运算的结果，同样成为下个步骤的输入：

   ```
   Franklin, Ben:OSD212:555-0022
   Ross, Betsy:BMD17:555-0033
   ...
   ```

4. 数据重新排序。数据之前已按照登录名称排序完成，但现在我们要的是以个人的姓来排序，这里使用 sort 命令：

   ```
   ... | sort -t: -k1,1 -k2,2 -k3,3 | ...
   ```

   这条命令是以冒号分隔字段，依次对字段 1、2 与 3 进行排序。运算的结果是下一步骤的输入：

   ```
   Franklin, Ben:OSD212:555-0022
   Gale, Dorothy:KNS321:555-0044
   ...
   ```

5. 最后，重新格式化输出，使用 awk 的 printf 语句，配合制表字符（Tab）分隔每个字段。命令如下：

   ```
   ... | awk -F: '{ printf("%-39s\t%s\t%s\n", $1, $2, $3) }'
   ```

   为了灵活性以及将来能易于维护，格式化应该留到最后。一直到这里，所有内容都还只是任意长度的文本字符串。

这里是完整的管道：

```
join -t: $PERSON $OFFICE |
    join -t: - $TELEPHONE |
```

```
        cut -d: -f 2- |
            sort -t: -k1,1 -k2,2 -k3,3 |
                awk -F: '{ printf("%-39s\t%s\t%s\n", $1, $2, $3) }'
```

awk的printf语句在这里的用法有点类似Shell的printf命令：用冒号作为分隔字符，显示第一个字段，并让字段1显示来的结果固定为向左对齐的39个字符长度，紧接着一个制表字符（Tab），再接着第二个字段，加上另一个制表字符，再接上第三个字段，完整结果如下：

```
Franklin, Ben                          •OSD212•555-0022
Gale, Dorothy                          •KNS321•555-0044
Gale, Toto                             •KNS322•555-0045
Hancock, John                          •SIG435•555-0099
Jefferson, Thomas                      •BMD19•555-0095
Jones, Adrian W.                       •OSD211•555-0123
Ross, Betsy                            •BMD17•555-0033
Washington, George                     •BST999•555-0001
```

所有的操作都已完成！整个脚本总共用了20多行（不含注释），即已包括了五个主要处理步骤。这些过程整理后如例5-1所示。

例5-1：建立办公室名录

```
#! /bin/sh
# 过滤/etc/passwd这类格式的输入流，
# 并以此数据衍生出办公室名录。
#
# 语法：
#       passwd-to-directory < /etc/passwd > office-directory-file
#       ypcat passwd | passwd-to-directory > office-directory-file
#       niscat passwd.org_dir | passwd-to-directory > office-directory-file

umask 077

PERSON=/tmp/pd.key.person.$$
OFFICE=/tmp/pd.key.office.$$
TELEPHONE=/tmp/pd.key.telephone.$$
USER=/tmp/pd.key.user.$$

trap "exit 1"                                HUP INT PIPE QUIT TERM
trap "rm -f $PERSON $OFFICE $TELEPHONE $USER" EXIT

awk -F: '{ print $1 ":" $5 }' > $USER

sed -e 's=/.*==' \
    -e 's=^\([^:]*\):\(.*\) \([^ ]*\)=\1:\3, \2=' < $USER | sort > $PERSON

sed -e 's=^\([^:]*\):[^/]*/\([^/]*\)/.*$=\1:\2=' < $USER | sort > $OFFICE

sed -e 's=^\([^:]*\):[^/]*/[^/]*/\([^/]*\)=\1:\2=' < $USER | sort > $TELEPHONE

join -t: $PERSON $OFFICE |
    join -t: - $TELEPHONE |
```

```
        cut -d: -f 2- |
            sort -t: -k1,1 -k2,2 -k3,3 |
                awk -F: '{ printf("%-39s\t%s\t%s\n", $1, $2, $3) }'
```

Shell 脚本真正好用的地方是：当我们想修改脚本让它做点不一样的事，例如插入由 *key:jobtitle* 文件而来的工作职称时，只需要修改最后的管道，如下所示：

```
join -t: $PERSON /etc/passwd.job-title |              用来取出工作职称的join
    join -t: - $OFFICE |
        join -t: - $TELEPHONE |
            cut -d: -f 2- |
                sort -t: -k1,1 -k3,3 -k4,4 |          修改sort命令
                    awk -F: '{ printf("%-39s\t%-23s\t%s\t%s\n",
                              $1, $2, $3, $4) }'      格式化命令
```

多加一个额外的名录字段的总成本是多一个join、更改sort字段以及调整最后的awk格式化命令。

由于我们小心地在输出结果上保留特殊的字段定界符，因此可以不留痕迹地准备好替代的名录，如下所示：

```
passwd-to-directory < /etc/passwd | sort -t'•' -k2,2 > dir.by-office
passwd-to-directory < /etc/passwd | sort -t'•' -k3,3 > dir.by-telephone
```

如前所述：• 表示 ASCII 的制表字符。

在此程序里，重要的假设是在每条数据记录的一个唯一键值 (unique key)。有了这个唯一键值，数据的各种不同视图可以用成对的*key:value*方式维护在文件中。这里的键值为 UNIX 的用户名称，但在较大型的例子中，键值很有可能是书目编号 (ISBN)、信用卡号码、员工编号、国家退休体系编号、产品序号、学号等。现在你终于知道我们身上有多少编号了吧！有时在处理这些数据时需要的不一定是号码：只是需要具唯一值的文本字符串。

5.2 针对 Web 的结构型数据

由于 World Wide Web (WWW) 广为流行，所以在前一节中开发办公室名录的形式，可以稍作修改，让数据以较漂亮的形式呈现。

Web 文件多半都是由 *Hyper Text Markup Language (HTML)* 语言写成；它是 *Standard Generalized Markup Language (SGML)* 家族语言之一，而 SGML 自 1986 年起，陆续被定义在数个 ISO 标准中。本书的原稿是用 DocBook/XML 写成，它也是 SGML 的一个

特定实例。如果你有兴趣，可参考《HTML in HTML & XHTML: The Definitive Guide》(O'Reilly)，该书对 HTML 有完整介绍（注 2）。

数据库小记

现今许多商用数据库都以“关系数据库”构建：数据可以以一对 *key:value* 形式访问，并结合（join）操作用于构成多栏表格，提供选定的数据子集的视图。关系数据库是由E.F.Codd[a]在1970年首度提出，尽管数据库业界初始反对，认为关系数据库无法有效率地实现出来，不过Codd仍积极地推动着。幸好，聪明的程序设计师们马上就找到了解决效率问题的方法。Codd 功不可没，在 1981 年他拿到了 ACM 图灵奖，这个奖项被誉为计算机科学领域里的诺贝尔奖。

时至今日，陆续出现许多 *Structured Query Language*（结构化查询语言）的 ISO 标准，让独立于厂商的数据库可以被访问，而这其中最重要的一个 SQL 操作，就是`join`。讨论SQL 的书已经很多了，若你想再进一步了解，可以挑一本通用的书参考，例如《SQL in a NutShell》[b]。我们的简易办公室名录操作也包含了现代关系数据库核心概念的重要课题以及UNIX软件工具，在准备向数据库中输入数据并处理它们的输出时，这些工具也是很有用的。

a: E.F.Codd,《A Relational Model of Data for Large Shared Data Banks》, Communications of the ACM, 13(6) 377-387, June (1970), 及 Relational Database:《A Practical Foundation for Productivity》, Communications of the ACM, 25(2) 109-117, February(1982)(Turing Award 讲座)。

b: Kevin Kline 与 Daniel Kline 合著，O'Reilly & Associates 出版，ISBN 1-56592-744-3。其他 SQL 相关书籍列表也可参考 *http://www.math.utah.edu/pub/tex/bib/sqlbooks.html*。

我们在这个小节，只需要小型的HTML子集，这部分我们将用一小段文字来介绍。如果你对 HTML 已熟悉，可以跳过这两页。

下面是我们写的一个遵循标准的小型HTML文件，是由我们其中一人所编写的一个好用工具所产生的（注 3）：

```
$ echo Hello, world. | html-pretty
<!-- -*-html-*- -->
```

注 2： 除该书外（已列于书后的参考书目），还有许多 SGML 与其衍生产物的书列于 *http://www.math.utah.edu/pub/tex/bib/sgml.html* 与 *http://www.math.utah.edu/pub/tex/bib/sgml2000.html*，可供读者参考。

注 3： 见 *http://www.math.utah.edu/pub/sgml/*。

```
<!-- Prettyprinted by html-pretty flex version 1.01 [25-Aug-2001] -->
<!-- on Wed Jan  8 12:12:42 2003 -->
<!-- for Adrian W. Jones (jones@example.com) -->

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
<HTML>
    <HEAD>
        <TITLE>
            <!-- Please supply a descriptive title here -->
        </TITLE>
        <!-- Please supply a correct e-mail address here -->
        <LINK REV="made" HREF="mailto:jones@example.com">
    </HEAD>
    <BODY>
        Hello, world.
    </BODY>
</HTML>
```

在这个 HTML 输出中，值得注意的事项如下：

- HTML 以 <!-- 与 --> 括住注释。
- 特殊处理器命令包含在<!与>之中：这里 DOCTYPE 命令是告诉 SGML 子句解析器文件类型是什么，以及去哪里寻找其语法文件。
- 填写在尖括号里的标记字组，叫做标签（tag）。在 HTML 里，标签名称里的字母大小写不重要：html-pretty 将标签里的字母全都设置为大写，是为了便于阅读。
- 标记（markup）语言写的代码包括一个起始标签 <*NAME*> 与一个结束标签 </*NAME*>，且对许多标签而言，可嵌套设置，只要遵循 HTML 语法里定义的规则即可。
- HTML 文件是以包含一个 HEAD 与一个 BODY 的 HTML 对象构建而成。
- 在 HEAD 里，TITLE 对象定义的是文件标题，也就是显示在浏览器窗口标题栏上的那个，也是书签列表的默认名称。再者，HEAD 里，还有一个 LINK 对象，这多半是用来给出网页维护者的相关信息。
- 这个文件里，浏览器显示可看得见的范围则是 BODY 的内容。
- 引号括起来字符串以外的空格不重要，所以我们可以自由地使用垂直与水平的空格以完整地呈现所要强调的结构，就像 HTML 的 prettyprinter 所做的那样。
- 所有其他内容都是可打印的 ASCII 文字，只有三个例外。字面上的尖括号必须以特殊编码体现，叫做实体（entities），它包括 & 符号、标识符（identifier）以及分号，例如：<与 >。因为 & 已用来作为实体的起始字母，所以它自身有字面上的实体名称：&。针对重音字符，HTML 支持一个具有最现代的所有功能的实体，涵盖了大部分我们可以书写的西欧语言，例如，café du bon goût 会得到 café du bon goût。

- 虽然我们的小型范例里未提及，字体的修改也是可以在HTML里做到，可使用B（粗体）、EM（音节强调）、I（斜体）、STRONG（特粗）以及TT（打字机字体（固定宽度字符））环境，当你写<B>bold phrase</B>，会得到**bold phrase**。

要将我们的办公室名录转换成正式的HTML，只需要再知道一件事：如何格式化表格，因为那才是真正的办公室名录，且我们不想使用打字机字体，强制每一行在浏览器上显示时排列一致完全对齐。

在HTML 3.0及之后的版本上，表格包括一个TABLE环境，其内有行，每行是一个表格行（TR）环境。在每一行里是单元格（cell），叫做表格数据，每个单元格是一个TD环境。特别值得一提的是，每列（垂直）数据不接收任何特殊标记：一个数据列是一组单元格，取自表格里所有行（水平）中相同的行位置。幸好，我们无须预先声明行与列的数目。浏览器与格式化程序的工作便是收集所有的单元格，知道在每一列内的最宽单元格，再将表格格式化，使其列宽够大，足以能够容纳那些最宽的单元格。

在我们的办公室名录范例中，只需要三列，所以标记范例如下：

```
<TABLE>
    ...
    <TR>
        <TD>
            Jones, Adrian W.
        </TD>
        <TD>
            555-0123
        </TD>
        <TD>
            OSD211
        </TD>
    </TR>
    ...
</TABLE>
```

另一种等效的但较复杂也较难阅读的方式为：

```
<TABLE>
    ...
    <TR><TD>Jones, Adrian W.</TD><TD>555-0123</TD><TD>OSD211</TD></TR>
    ...
</TABLE>
```

因为我们选择保留办公室名录纯文本版里的特殊字段分隔字符，所以有足够的信息可以识别每列中的单元格。而且，因为HTML文件里，空白多半不带特殊含义，我们就不需要特别注意标签是否完好排列。如果之后有需要，html-pretty还是可以做得很完美。我们的转换过滤器有三个步骤：

1. 输出前置的样板文件（boilerplate）直到内文开始处。
2. 将名录里的每一行包括在表格标记里。
3. 输出结尾的样板文件（boilerplate）。

我们得在这个小范例里作点小变动：将 DOCTYPE 命令更新为近期版本的语法层级，看起来就像这样：

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN//3.0">
```

你无须记住这个，因为 html-pretty 有选项可产生任何标准 HTML 语法层级的输出，所以你只要从它的输出中，复制适用的 DOCTYPE 命令即可。

至此，显然大部分的工作只是在编写样板文件（boilerplate），不过这很简单，因为我们可以从小HTML范例中复制文字。唯一比较侧重编程的步骤是在中间部分，这部分只需几行 awk 就可以办到。不过，使用 sed 流编辑程序的替换功能，可以更简化工作，需要两个编辑命令：一个是以 </TD><TD> 取代嵌入的制表符定界符，另一个是将整行包括在 <TR><TD>...</TD></TR> 中。我们先临时假设名录里没有重音字符，不过就算要将尖括号与 & 符号加到输入流里也不难，只要增加三个初始的 sed 步骤即可。我们将完整的程序集放在例 5-2 中。

例 5-2：将办公室名录转换为 HTML 格式

```
#! /bin/sh
# 将制表符 (Tab) 所分隔的文件，转换为遵循语法的 HTML
#
# 用法：
#     tsv-to-html < infile > outfile

cat << EOFILE                                开头的样板文件 (boilerplate)
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN//3.0">
<HTML>
    <HEAD>
        <TITLE>
            Office directory
        </TITLE>
        <LINK REV="made" HREF="mailto:$USER@`hostname`">
    </HEAD>
    <BODY>
        <TABLE>
EOFILE

sed -e 's=&=\&amp;=g' \                      将特殊字符转换为实体 (entities)
    -e 's=<=\&lt;=g' \
    -e 's=>=\&gt;=g' \
    -e 's=\t=</TD><TD>=g' \                  提供表格标记 (markup)
    -e 's=^.*$=          <TR><TD>&</TD></TR>='
```

```
cat << EOFILE                              结尾的样板文件 (boilerplate)
        </TABLE>
    </BODY>
</HTML>
EOFILE
```

<<标记称为嵌入文件 (here document)。这点在7.3.1节里有较详细的解说。简单地讲，Shell读取所有行，直到接在<<之后的定界符为止（在本例是`EOFILE`）、在被包含的行上执行变量与命令替换，以及将结果当成标准输入给命令。

例5-2的脚本里很重要的一点就是：表格中的列（垂直）数是完全独立的！即它可以用来将任何以制表字符（Tab）分隔的文件转换成HTML。电子表格程序通常会以这样的格式存储数据，所以我们的简单工具，可以将电子表格数据转换产生正确的HTML。

我们对于`tsv-to-html`很小心，是为了维护原始办公室名录的空格结构，因为这么一来我们往下应用更进一步的过滤器时会很轻松。没错，`html-pretty`正是为此而编写的。HTML标记配置的标准化，大大地简化了其他HTML工具程序的工作。

那么，我们要如何将重音字符转换为HTML实体呢？我们当然可以加入额外的编辑步骤来扩展`sed`命令，如加入`-e 's=é=é=g'`。可是，这里有100个左右的实体要满足，而且，当需要将其他形式的文本文件转换成HTML时，我们可能也会需要相似的替换。

这就是为什么应该将这样的工作分给另一个程序来做，让我们日后能再重复使用，无论它是作为管道步骤，紧接着例5-2的`sed`命令，还是作为稍后要应用的过滤器（这是“转而构建专门的工具”原则）。这样的程序只是一个含有替换命令的冗长乏味的列表，且我们需要每个本地文本内码的相对信息，例如各种的ISO 8895-*n* 内码页（code page），这部分在附录B中已有介绍。我们不在这里完整介绍过滤器，不过在例5-3的代码片段中，你应该能大致窥见其模式。HTML的实体库并不足够应付其他重音字符，但由于World Wide Web的趋势是正以Unicode与XML取代之前的ASCII与HTML，通过消除字符集限制，会有不同的方法解决这个问题。

例5-3：iso8859-1-to-html 程序片段

```
#! /bin/sh
# 将输入流里内含符合 ISO 8859-1 编码且范围在 128 .. 255 的字符，
# 转换为 HTML 对等的 ASCII。
# 字符 0 .. 127 保留作为一般的 ASCII。
#
# 语法：
#       iso8859-1-to-html infile(s) >outfile

sed \
        -e 's= =\ =g' \
        -e 's=¡=\&iexcl;=g' \
```

```
        -e 's=¢=\&cent;=g' \
        -e 's=£=\&pound;=g' \
...
        -e 's=ü=\&uuml;=g' \
        -e 's=ý=\&yacute;=g' \
        -e 's=þ=\&thorn;=g' \
        -e 's=ÿ=\&yuml;=g' \
        "$@"
```

这个过滤器的使用方式如下：

```
$ cat danish                                    显示ISO 8859-1编码的Danish范例文字
Øen med åen lå i læ af én halvø,
og én storØ, langs den græske kyst.

$ iso8859-1-to-html danish                      将文字转换为HTML实体
Øen med åen lå i lå af én halvø,
og én stor ø, langs den græske kyst.
```

5.3 文字解谜好帮手

字谜游戏会给你一些单词的线索，但大部分时候我们还是被困住，例如：具有10个字母的单词，以a b起始，且第七个字不是x就是z。

用awk或grep进行正则表达式模式匹配是必需的，问题是：要查找什么文件呢？使用UNIX拼写字典是不错的选择，大部分系统的/usr/dict/words下都应该找得到它（还有像/usr/share/dict/words与/usr/share/lib/dict/words也是可能出现的地方）。这是一个简单的文本文件，每行一个单词，以字典顺序排列。我们可以轻松地从任何的文本文件集合建立另一个具相似外表的文件，如下所示：

```
cat file(s) | tr A-Z a-z | tr -c a-z\' '\n' | sort -u
```

第二个管道步骤是将大写字母转换成小写，第三个则是以换行字符取代非字母字符，最后为结果进行排序，并去除重复部分，让每行都为唯一值。在第三步里，视撇号（'）为字母，因为它们在缩写里会用到。每个UNIX系统都具有可以此方式处理的整组文字－例如格式化后的手册页在/usr/man/cat*/*与/usr/local/man/cat*/*内。我们的系统里就有一个提供了一百万行以上的文本，并且产生了44 000个左右的唯一单词。在Internet上你也可以找到很多种语言的单词列表（注4）。

注4： 可在*ftp://ftp.ox.ac.uk/pub/wordlists/*、*ftp://qiclab.scn.rain.com/pub/wordlists/*、*ftp://ibiblio.org/pub/docs/books/gutenberg/etext96/pgw**，以及*http://www.phreak.org/html/wordlists.shtml*中取得。也可以直接在Internet上查找“word list”，一样能找到很多相关信息。

我们假设已经以此方式建立了单词列表的集合，并将它们存储在一个标准的地方，以便可以从脚本中参考到它。我们编写的程序如例 5-4 所示。

例 5-4：文字解谜的好帮手

```
#! /bin/sh
# 通过一堆单词列表，进行类似egrep(1)的模式匹配
# word lists
#
# 语法：
#       puzzle-help egrep-pattern [word-list-files]

FILES="
        /usr/dict/words
        /usr/share/dict/words
        /usr/share/lib/dict/words
        /usr/local/share/dict/words.biology
        /usr/local/share/dict/words.chemistry
        /usr/local/share/dict/words.general
        /usr/local/share/dict/words.knuth
        /usr/local/share/dict/words.latin
        /usr/local/share/dict/words.manpages
        /usr/local/share/dict/words.mathematics
        /usr/local/share/dict/words.physics
        /usr/local/share/dict/words.roget
        /usr/local/share/dict/words.sciences
        /usr/local/share/dict/words.UNIX
        /usr/local/share/dict/words.webster
      "
pattern="$1"

egrep -h -i "$pattern" $FILES 2> /dev/null | sort -u -f
```

FILES变量保存了单词列表文件的内建列表，可供各个本地站点定制。grep的-h选项指示最后结果不要显示文件名，-i选项为忽略字母大小写，我们还用了2> /dev/null丢弃标准错误信息的输出，这是用于单词列表文件不存在或是在它们缺乏必需的读取权限的情况（这种重定向在7.3.2节里有详尽的介绍）。最后的sort步骤则可以简化最后的结果，让列表里没有重复单词，并忽略字母大小写。

现在就可以找到我们要寻找的单词了：

```
$ puzzle-help '^b.....[xz]...$' | fmt
bamboozled Bamboozler bamboozles bdDenizens bdWheezing Belshazzar
botanizing Brontozoum Bucholzite bulldozing
```

能找出每行有 6 个辅音字母的英文单词吗？你可以这么做：

```
$ puzzle-help '[^aeiouy]{6}' /usr/dict/words
Knightsbridge
mightn't
oughtn't
```

若你觉得*y*不算是元音，那就会显示更多的单词，例如*encryption*、*klystron*、*porphyry*、*syzygy*都是。

只要在最后加上过滤步骤：`egrep -i '^[a-z]+$'`，我们可以迅速排除列表里有缩写的那些，不过它们出现在单词列表里无伤大雅。

5.4 单词列表

在1983年到1987年之间，贝尔实验室的研究人员Jon Bentley在《Communications of the ACM》写了篇有趣的专栏《Programming Pearls》。专栏的部分文章集结之后，作了相当程度的变动，作为两本书出版 —— 见本书后参考书目列表。专栏中有篇文章是Bentley下的战帖：写一个文字处理程序，找出*n*个出现最频繁的单词，并在输出结果的列表上加入它们出现的次数，按照次数由大至小排序。著名计算机科学家Donald Knuth与David Hanson分别回应了两个聪明有趣的程序（注5），每个程序都是花上数小时编写出来的。Bentley最初的定义并不精准，因而Hanson再次给出一个解释：在给定的文本文件以及整数*n*下，必须将单词显示出来（还要加上它们出现频率），按照这些单词出现的频率，从出现最多的*n*次，依次往下排列。

针对Bentley的第一篇文章，贝尔实验室的研究人员Doug McIlroy回头检查Knuth的程序，提供一个6个步骤UNIX解决方案，仅需几分钟便能开发完成，且第一次就运行无误。此外，不同于其他两个程序的地方是：McIlroy的程序并未指定限制性的常量，包括单词长度、唯一单词的数目以及输入文件大小。也就是说，它的想法是：一个单词的构建完全是由一个简单的模式所定义，这在他程序最前面的两行可执行语句里给定，这使得单词识别算法的更改变得容易了。

McIlroy的程序阐明了UNIX工具程序的处理方式是强大的：将复杂的问题切分成数个较简单的部分，简单到你已经知道这个部分该怎么处理。为解决单词出现频率问题，McIlroy将纯文本文件转换为单词列表，一行一个字（由`tr`来完成此工作）、将单词对应到单一的字母大小写（一样还是用`tr`）、单词列表的排序（用`sort`）、从单词列表中去除重复部分，简化为仅提供唯一的单词（用`uniq`）且加上计数、再将单词列表以计数的数字由大至小排序，最后，显示单词列表的前几项（这里是使用`sed`，不过`head`也可以）。

注5：《Programming Pearls: A Literate Program》: A WEB program for common words, Comm.ACM 29(6),471-483, June(1986)；以及《Programming Pearls: Literate Programming: Printing Common Words》, 30(7), 594-599, July(1987)。Knuth的论文也再版于《Literate Programming》书中，由Stanford University Center for the Study of Language and Information出版：1992年，ISBN 0-937073-80-6（平装版）与0-937073-81-4（精装版）。

最后形成的程序应该值得给它个名字 wf（word frequency，单词出现频率之意），然后附上注释标题，打包成 Shell 脚本。我们也扩充了 McIlroy 原本的 sed 命令，让输出列表长度参数可选，并现代化 sort 选项。完整程序可见例 5-5。

例 5-5：单词出现频率过滤器

```
#! /bin/sh
# 从标准输入读取文本流，再输出出现频率最高的前 n（默认值：25）个单词的列表
# 附上出现频率的计数，按照这个计数由大而小排列，
# 输出到标准输出。
#
# 语法 ：
#       wf [n]

tr -cs A-Za-z\' '\n' |                将非字母字符置换成换行符号
  tr A-Z a-z |                        所有大写字母转为小写
    sort |                            由小而大排序单词
      uniq -c |                       去除重复，并显示其计数
        sort -k1,1nr -k2 |            计数由大而小排序后，再按照单词由小而大排序
          sed ${1:-25}q               显示前 n 行（默认为 25）；见第 3 章
```

POSIX 的 tr 支持 ISO Standard C 的所有转义序列（escape sequences）。较旧的 X/Open Portability Guide 规格则仅有八进制的转义序列，且原始的 tr 更是完全不支持，会强制将换行符号逐字写出，这也是 McIlroy 原始程序里被批判的一点。还好，我们测试过的所有系统，其 tr 命令现在都支持 POSIX 转义序列。

Shell 管道并不是使用 UNIX 工具解决此问题的唯一方式：Bentley 提出的 awk 程序实作只有 6 行，在他早期的专栏里已出现过了（注 6），内容上大致与 McIlroy 的管道之前 4 个步骤差不多。

Knuth 与 Hanson 讨论了他们的程序在计算上的复杂度，且 Hanson 使用运行时探测（runtime profiling）的方式研究该程序的数种变化，以找出最快的一种。

McIlroy 的复杂度是容易识别的。sort 之外的所有步骤，执行时间都与其输入数据的大小呈线性关系，输入量多半在 uniq 步骤之后会明显减少。因此，会牵制速度的其实是第一步：sort。一个好的排序算法主要是看它的比较功能，以 UNIX 的 sort 来看，排序 n 个项目的时间是与 $n \log_2 n$ 成正比。这个底为 2 的对数值会很小：假设 n 为一百万，则它就是大约 20。因此，在实际上，我们预期 wf 会比只是使用 cat 复制它的输入流还要慢些。

注 6：《Programming Pearls: Associative Arrays》, Comm. ACM 28(6), 570-576, June, (1985)。这是一篇介绍关联式数组（associative arrays）的好文章（表格是通过字符串来索引，而不是整数），这是大多数脚本语言的常见功能。

这里用莎士比亚的作品来套用这个脚本试试，我们用最著名的《哈姆雷特》（注7），使用pr重新格式化输出的结果，以每行4列显示：

```
$ wf 12 < hamlet | pr -c4 -t -w80
  1148 the          671 of           550 a            451 in
   970 and          635 i            514 my           419 it
   771 to           554 you          494 hamlet       407 that
```

结果大致和一般英语散文的预期相同，不过更好玩的是，我们可以要求算出去除掉重复字后有多少单词出现在此剧中：

```
$ wf 999999 < hamlet | wc -l
4548
```

再看看最不常出现的字有哪些，仅显示一部分：

```
$ wf 999999 < hamlet | tail -n 12 | pr -c4 -t -w80
      1 yaw          1 yesterday         1 yielding          1 younger
      1 yawn         1 yesternight       1 yon               1 yourselves
      1 yeoman       1 yesty             1 yond              1 zone
```

999999这个参数值没有任何意义，我们只是要一个很大的数字而已，几乎可以确定大于单词计数的数字，再辅以键盘的重复功能，使得这个参数值很容易输入。

我们还可以要求算出这4548个单词里有几个是只出现一次的：

```
$ wf 999999 < hamlet | grep -c '^ *1·'
2634
```

在grep模式中接在数字1后面的•表示的是制表字符（Tab）。这个结果有点令人意外，对于现在的英语散文来说可能不常是这样：虽然这个剧本的词汇很多，但接近58 %的字都只出现过一次？！然而，经常出现的几个核心单词则相当的少：

```
$ wf 999999 < hamlet | awk '$1 >= 5' | wc -l
740
```

这大约是一个学生在外语课程里一学期学到的单词数，或是学龄前儿童会的单词数。

莎士比亚没有计算机帮他分析他的作品（注8），但我们可以料想的是，以他的天才，写出一篇让大众都可理解的文章是轻而易举的。

注7：　你可以在*http://www.gutengerg.net/*找到该剧，Project Gutenberg是很不错的文件宝藏库。

注8：　的确，他的作品里只有一个计算机的起源字："computation"，而且仅在两个剧本《Comedy of Errors》与《King Richard III》里各出现一次。而"Arithmetic"出现了6次，"calculate"出现了2次，"mathematics"出现了3次。

我们将wf套用到莎士比亚的各出剧里，发现《哈姆雷特》(Hamlet)用了最多单词(4548)，而《难得糊涂》(Comedy of Errors) 则是最少的。莎士比亚全集里 —— 含剧本与十四行诗，所有不重复单词加起来就有将近23700个，说明你需要浸淫在许多戏剧中才能享受它们的丰富性。大约有 36% 的单词只用过一次，而只有一个字是以 x 开头的：即 Xanthippe，这是出现在《驯悍记》(Taming of the Shrew) 里。无疑，莎士比亚的作品不但为字谜游戏提供了一个丰沛的来源，也是词汇分析学家很好的研究题材！

5.5 标签列表

tr命令可用于取得单词列表，但其更常见的用法是：将一组字符集转换成另外一组，就像我们在前一节例 5-5 介绍的那样，这是一个值得记住的方便好用的 UNIX 工具。这也导致了一个问题，也就是我们在写这本书时遇到的：如何确保整篇 5 万行左右的原稿文件具有一致的标记 (markup) 呢？例如，当我们在正文中提到一条命令时，可将命令标记为 <command>tr</command>，但是在其他地方，我们可能举例说明你输入的内容，便会用<literal>tr</literal>这样的标记。还有一种是提及手册页参考时，标记形式为 <emphasis>tr</emphasis>(1)。

例 5-6 里的 taglist 程序就是这类问题的解决方案。该程序找出写在同一行里开始 / 结束的一对标签 (tag)，然后再输出一个排序列表，该列表将标签的使用与输入文件相关联。此外，对于多次的方式标记相同单词的地方，给出一个箭头标志。下列片段即为本章内容应用该程序后的输出：

```
$ taglist ch05.xml
...
      2 cut                              command         ch05.xml
      1 cut                              emphasis        ch05.xml <----
...
      2 uniq                             command         ch05.xml
      1 uniq                             emphasis        ch05.xml <----
      1 vfstab                           filename        ch05.xml
...
```

列出标签的任务想当然是很复杂的，且以最传统的程序语言来做也有点难，即使像 Java 和 C 这样拥有大规模类库，并且即便从 Knuth 或 Hanson 处理单词频率问题的程序开始也一样。但使用 UNIX 的管道，搭配你已熟悉的几个工具，只要 9 个步骤就能完成。

单词出现频率计算程序无法处理多个命名的文件：它假设仅有单一流。不过这也不是太严重的限制，因为我们可以很简单地通过 cat 将多个输入文件喂给它。不过在这里，我们需要有文件名，因为知道出问题但不晓得问题出在哪里，这对我们是没有好处的。所以，文件名成了 taglist 的单个参数，在脚本中可通过 $1 取得。

1. 我们通过cat将输入文件给管道。当然，也可以省掉这个步骤，只需从$1中重定向下一步骤的输入，不过我们觉得在一个复杂的管道里将“数据生成”与“数据处理”分开，会比较有条理，而这么做在程序日后需要在另一个步骤插入新的管道时，也会容易些。

```
cat "$1" | ...
```

2. 通过sed简化Web URL所需的另一种复杂标记：

```
... | sed -e 's#systemitem *role="url"#URL#g' \
          -e 's#/systemitem#/URL#' | ...
```

这是将标签，如<systemitem role="URL">与</systemitem>，分别转换成较简单的<URL>与</URL>标签。

3. 下一个步骤使用tr将空白与成对的定界符转换为换行符（newline）：

```
... | tr ' (){}[]' '\n\n\n\n\n\n\n' | ...
```

4. 至此，输入数据包括一行一个单词（word）（或是空行）。这里所指的单词，不是真正的文本就是SGML/XML标签。所以下一步骤我们使用egrep，选定由标签括起来的单词：

```
... | egrep '>[^<>]+</' | ...
```

这个正则表达式会匹配由标签括住的单词：右尖括号，接着至少一个非尖括号字符，跟着一个左尖括号，再接上一个斜杠（也就是结束标签）。

5. 至此，输入数据包括了带有标签的行。第一个awk步骤使用尖括号作为字段分隔字符，所以当输入为<literal>tr</literal>，即切分为4栏，依次是：一个空字段、literal、tr，最后为/literal。文件名通过命令行传给awk，其中-v选项把awk变量FILE设置为此文件名。该变量之后会应用到print语句上，以输出单词、标签与文件名：

```
... | awk -F'[<>]' -v FILE="$1" \
          '{ printf("%-31s\t%-15s\t%s\n", $3, $2, FILE) }' | ...
```

6. sort步骤是以单词顺序排列每一行：

```
... | sort | ...
```

7. uniq命令提供初始的计数字段。输出为记录列表，其中字段依次为计数、单词、标签、文件：

```
... | uniq -c | ...
```

8. 第二个sort是将输出结果以单词及标签的顺序排列（第2、3字段）：

```
... | sort -k2,2 -k3,3 | ...
```

9. 最后步骤是使用一个小小的awk程序，过滤掉连续的行，加上结尾的箭头符号，当出现与上一行相同单词时使用。然后，此箭头符号可清楚地指出哪个字使用了不同的标记，也就是作者、编辑或出版社相关人员应特别检查的地方：

```
... | awk '{
                print ($2 == Last) ? ($0 " <----") : $0
                Last = $2
            }'
```

完整的程序如例 5-6 所示。

例 5-6：产生 SGML 标签列表

```
#! /bin/sh -
#       读取命令行上给定的HTML/SGML/XML文件，
#       找出包含像<tag>word</tag>这样的标记，再输出到标准输出，
#       该标准输出将以制表字符（tab）分隔字段，依次为
#
#       计数    单词    标签    文件名
#       按照单词与标签由小至大排序。
#
# 语法 :
#        taglist xml-file

cat "$1" |
  sed -e 's#systemitem *role="url"#URL#g' -e 's#/systemitem#/URL#' |
    tr ' (){}[]' '\n\n\n\n\n\n\n' |
      egrep '>[^<>]+</' |
        awk -F'[<>]' -v FILE="$1" \
            '{ printf("%-31s\t%-15s\t%s\n", $3, $2, FILE) }' |
          sort |
            uniq -c |
              sort -k2,2 -k3,3 |
                awk '{
                        print ($2 == Last) ? ($0 " <----") : $0
                        Last = $2
                    }'
```

在 6.5 节里，我们将告诉你如何将标签列表的运算应用到多文件的情况上。

5.6 小结

本章说明的是解决许多文字处理问题的一些方法，这些问题里没有一个是能够在大部分程序语言中简单地解决的。本章主要话题如下：

- 数据标记相当有价值，但又不能太复杂。具有唯一性的单一字符，例如制表字符（Tab）、冒号或是逗点通常就够用了。
- 将单纯的 UNIX 工具与管道结合使用，加上在适合的文字处理语言中的简短程序——例如 awk，可灵活运用数据标记来传递多个数据片段，使其通过一系列的处理步骤，产生有用的报告。

- 通过保持数据标记的简单，我们的工具所产生的输出，可以马上变成新工具的输入，就像单词频率次数过滤器（wf 那样的输出分析）可应用到莎士比亚的作品上一样。
- 将一些小型标记保留在输出结果里，之后还能再进一步处理这样的数据，就像我们把简单的ASCII办公室名录转换成网页形式那样。绝不要以为任何一种电子数据的形式就会是最后的结果：现在有越来越多能将四页显示成一面的程序语言，例如PCL、PDF以及PostScript，它们可以保留原始标记，再进行页面的格式化。字处理程序所产生的文件现在也缺乏有条理的标记方式，不过这种情况即将转变！就在写这本书的同时，已有优秀的字处理程序厂商提出："正考虑将XML表达式作为文件的存储格式"。GNU 项目的 gnumeric 电子表格程序、Linux Documentation Project（注 9）以及 OpenOffice.org（注 10）的办公软件都已经这么做了。
- 以定界符分隔字段的行，这是一种方便用来与更复杂软件交换数据的格式，例如电子表格与数据库。不过这类系统通常都提供某种形式的报表产生功能，可轻易地将数据提取为分栏的流，之后便能利用适当的程序语言所写的过滤器，更进一步地处理这些数据。例如产品目录与名录发布都是这种方法的最佳应用。

注 9： 见 *http://www.tldp.org/*。

注 10： 见 *http://www.openoffice.org/*。

第6章

变量、判断、重复动作

变量对于正规程序而言很重要。除了维护有用的值作为数据，变量还用于管理程序状态。由于Shell主要是字符串处理语言，所以你可以利用Shell变量对字符串值做很多事。然而，因为算术运算也是必要的，所以POSIX Shell也提供利用Shell变量执行算术运算的机制。

流程控制的功能造就了程序语言：如果你有的只是命令语句，是不可能完成任何工作的。本章介绍了用来测试结果、根据这些结果做出判断以及加入循环的功能。

最后介绍的是函数：它可以将相关工作的语句集中在同一处。这么一来就可以在脚本里的任何位置，轻松执行此工作。

6.1 变量与算术

Shell变量如同传统程序语言的变量一样，是用来保存某个值，直到你需要它们为止。我们在2.5.2节里已介绍过Shell变量名称与值的基本概念，但除此之外，Shell脚本与函数还有位置参数（positional parameter）的功能；传统的说法应该是“命令行参数”。

Shell脚本里经常出现一些简单的算术运算，例如每经过一次循环，变量就会加1。POSIX Shell为内嵌（inline）算术提供了一种标记法，称为算术展开（arithmetic expansion）。Shell会对`$((...))`里的算术表达式进行计算，再将计算后的结果放回到命令的文本内容。

6.1.1 变量赋值与环境

Shell变量的赋值与使用方式已在2.5.2节中提过，但这个小节将解释之前未提及的内容。

有两个相似的命令提供变量的管理，一个是readonly，它可以使变量成为只读模式；而赋值给它们是被禁止的。在Shell程序中，这是创建符号常量的一个好方法：

```
hours_per_day=24 seconds_per_hour=3600 days_per_week=7      赋值
readonly hours_per_day seconds_per_hour days_per_week       设为只读模式
```

export, readonly

语法

export *name*[=*word*] ...

export -p

readonly *name*[=*word*] ...

readonly -p

用途

export用于修改或打印环境变量，readonly则使得变量不得修改。

主要选项

-p

打印命令的名称以及所有被导出（只读）变量的名称与值，这种方式可使得Shell重新读取输出以便重新建立环境（只读设置）。

行为模式

使用-p选项，这两条命令都会分别地打印它们的名称以及被导出的或只读的所有变量与值。否则，会把适当的属性应用到指定的变量。

警告

许多商用UNIX系统里的/bin/sh，仍然不是POSIX兼容版本。因此，export与readonly的变量赋值形式可能无法工作。要实现最严格的可移植性，可使用：

```
FOO=somevalue
export FOO

BAR=anothervalue
readonly BAR
```

较常见的命令是export，其用法是将变量放进环境（environment）里。环境是一个名称与值的简单列表，可供所有执行中的程序使用。新的进程会从其父进程继承环境，也可以在建立新的子进程之前修改它。export命令可以将新变量添加到环境中：

```
PATH=$PATH:/usr/local/bin                  更新 PATH
export PATH                                导出它
```

最初的Bourne Shell会要求你使用一个两步骤的进程；也就是，将赋值与导出（export）

或只读（readonly）的操作分开（如前所示）。POSIX标准允许你将赋值与命令的操作结合在一起：

```
readonly hours_per_day=24 seconds_per_hour=3600 days_per_week=7

export PATH=$PATH:/usr/local/bin
```

export 命令可用于显示当前环境：

```
$ export -p                                     显示当前的环境
export CDPATH=":/home/tolstoy"
export DISPLAY=":0.0"
export ENV="/home/tolstoy/.kshrc"
export EXINIT="set ai sm"
export FCEDIT="vi"
...
```

变量可以添加到程序环境中，但是对 Shell 或接下来的命令不会一直有效：将该（变量）赋值，置于命令名称与参数前即可：

```
PATH=/bin:/usr/bin awk '...' file1 file2
```

这个PATH值的改变仅针对单个awk命令的执行。任何接下来的命令，所看到的都是在它们的环境中 PATH 的当前值。

export命令仅将变量加到环境中，如果你要从程序的环境中删除变量，则要用env命令，env 也可临时地改变环境变量值：

```
env -i PATH=$PATH HOME=$HOME LC_ALL=C awk '...' file1 file2
```

-i 选项是用来初始化（initializes）环境变量的；也就是丢弃任何的继承值，仅传递命令行上指定的变量给程序使用。

unset 命令从执行中的Shell中删除变量与函数。默认情况下，它会解除变量设置，也可以加上 -v 来完成：

```
unset full_name                                 删除 full_name 变量
unset -v first middle last                      删除其他变量
```

使用 unset -f 删除函数：

```
who_is_on () {                                  定义函数
    who | awk '{ print $1 }' | sort -u          产生排序后的用户列表
}
...
unset -f who_is_on                              删除函数
```

Shell早期版本没有函数功能或unset命令；POSIX加入了 -f 选项，以执行删除函数的操作，之后还加入 -v 选项，以便与 -f 相对应。

env

语法

`env [ -i ] [ var=value ... ] [ command_name [ arguments ... ] ]`

用途

当 *command_name* 被 env 执行时，可针对被 *command_name* 继承而来的环境有更细致的控制。

主要选项

-i

忽略继承的环境，仅使用命令行上所给定的变量与值。

行为

未提供 *command_name* 时，显示环境中所有变量的名称与其值。否则，在命令行上使用变量赋值，在引用 *command_name* 之前，以修改继承的环境。加上 -i 选项，env 会完全忽略继承的环境，且只使用所提供的变量与值。

警告

打印时，env 不会正确地为环境变量值加上引号，以供重新输入到 Shell 中。如果需要此功能，可使用 export -p。

unset

语法

`unset [ -v ] variable ...`

`unset -f function ...`

用途

从当前 Shell 删除变量与函数。

主要选项

-f

解除（删除）指定的函数。

-v

解除（删除）指定的变量。没有任何选项时，这是默认行为模式。

行为模式

如果没有提供选项，则参数将视为变量名称，并告知变量已删除。使用 -v 选项也会发生相同的行为。如使用 -f 选项，参数则被视为函数名称，并删除函数。

注意： myvar=赋值并不会将myvar删除，只不过是将其设为null字符串。相对地：unset myvar 则会完全删除它。这一差异在于“是变量设置”以及“是变量设置，但非null”展开，这部分将在下一个小节说明。

6.1.2 参数展开

参数展开（parameter expansion）是Shell提供变量值在程序中使用的过程；例如，作为给新变量的值，或是作为命令行的部分或全部参数。最简单的形式如下所示：

```
reminder="Time to go to the dentist!"     将值存储在reminder中
sleep 120                                 等待两分钟
echo $reminder                            显示信息
```

在Shell下，有更复杂的形式可用于更特殊的情况。这些形式都是将变量名称括在花括号里（${variable}），然后再增加额外的语法以告诉Shell该做些什么。花括号本身也是很好用的，当你需要在变量名称之后马上跟着一个可能会解释为名称的一部分的字符时，它就派得上用场了：

```
reminder="Time to go to the dentist!"     将值存储在reminder中
sleep 120                                 等待两小时
echo _${reminder}_                        加下划线符号强调显示的信息
```

警告： 默认情况下，未定义的变量会展开为null（空的）字符串。程序随便乱写，就可能会导致灾难发生：

```
rm -fr /$MYPROGRAM                 如未设置MYPROGRAM，就会有大灾难发生了！
```

所以，写程序要一直非常小心！

6.1.2.1 展开运算符

第一组字符串处理运算符用来测试变量的存在状态，且为在某种情况下允许默认值的替换。如表6-1所示。

表6-1：替换运算符

运算符	替换
${*varname*:-*word*}	如果*varname*存在且非null，则返回其值；否则，返回*word*。 **用途**：如果变量未定义，则返回默认值。 **范例**：如果count未定义，则${count:-0}的值为0。

表6-1：替换运算符（续）

运算符	替换
`${varname:=word}`	如果 *varname* 存在且不是 null，则返回它的值；否则，设置它为 *word*，并返回其值。 **用途**：如果变量未定义，则设置变量为默认值。 **范例**：如果 count 未被定义，则 ${count:=0} 设置 count 为 0。
`${varname:?message}`	如果 *varname* 存在且非 null，则返回它的值；否则，显示 *varname*:*message*，并退出当前的命令或脚本。省略 *message* 会出现默认信息 parameter null or not set。注意，在交互式 Shell 下不需要退出（在不同的 Shell 间会有不同的行为，用户需自行注意）。 **用途**：为了捕捉由于变量未定义所导致的错误。 **范例**：${count:?"undefined!"} 将显示 count: undefined!，且如果 count 未定义，则退出。
`${varname:+word}`	如果 *varname* 存在且非 null，则返回 *word*；否则，返回 null。 **用途**：为测试变量的存在。 **范例**：如果 count 已定义，则 ${count:+1} 返回 1（也就是"真"）。

表6-1里每个运算符内的冒号（:）都是可选的。如果省略冒号，则将每个定义中的"存在且非 null"部分改为"存在"，也就是说，运算符仅用于测试变量是否存在。

表6-1中的运算符已在 Bourne Shell 下使用了20多年。POSIX 标准化额外的运算符，用来执行模式匹配与删除变量值里的文本。新的模式匹配运算符，通常是用来切分路径名称的组成部分，例如目录前缀与文件名后缀。除了列出 Shell 的模式匹配运算符之外，表6-2 也展现了这些运算符的运行范例。在这些例子里，我们都假设变量 path 的值为 /home/tolstoy/mem/long.file.name。

注意：表6-2中运算符使用的模式，以及 Shell 里其他地方，例如 case 语句里所使用的模式，都为 Shell "通配字符（wildcard）"模式。这些在7.5节里有详细的说明。我们希望你能通过每天使用 Shell 来熟悉这些基本功能。

表6-2：模式匹配运算符

运算符	替换
`${variable#pattern}`	如果模式匹配于变量值的开头处，则删除匹配的最短部分，并返回剩下的部分。

表 6-2：模式匹配运算符（续）

运算符	替换
例：`${path#/*/}`	结果：`tolstoy/mem/long.file.name`
例：`${variable##pattern}`	如果模式匹配于变量值的开头处，则删除匹配的最长部分，并返回剩下的部分。
例：`${path##/*/}`	结果：`long.file.name`
例：`${variable%pattern}`	如果模式匹配于变量值的结尾处，则删除匹配的最短部分，并返回剩下的部分。
例：`${path%.*}`	结果：`/home/tolstoy/mem/long.file`
例：`${variable%%pattern}`	如果模式匹配于变量值的结尾处，则删除匹配的最长部分，并返回剩下的部分。
例：`${path%%.*}`	结果：`/home/tolstoy/mem/long`

这些看起来很难记，我们提供一个帮助记忆的好方法：# 匹配的是前面，因为数字正负号总是置于数字之前；% 匹配的是后面，因为百分比符号总是跟在数字的后面。另外一种帮助记忆的方式则是看传统的键盘配置（当然，指的是在美式键盘上）：# 位置靠左、% 靠右。

在这里用到的两种模式分别是：**/*/**，匹配任何位于两个斜杠之间的元素；**.***，匹配点号之后接着的任何元素。

最后，POSIX 标准化字符串长度运算符：`${#variable}` 返回 `$variable` 值里的字符长度：

```
$ x=supercalifragilisticexpialidocious     著名的特殊单词
$ echo There are ${#x} characters in $x
There are 34 characters in supercalifragilisticexpialidocious
```

6.1.2.2 位置参数

所谓位置参数（positional parameter），指的是 Shell 脚本的命令行参数（argument）；同时也表示在 Shell 函数内的函数参数。它们的名称是以单个的整数来命名。出于历史的原因，当这个整数大于 9 时，就应该以花括号（{}）括起来：

```
echo first arg is $1
echo tenth arg is ${10}
```

你也可以将前一节介绍的值测试与模式匹配运算符，应用到位置参数：

```
filename=${1:-/dev/tty}           如果给定参数则使用它，如无参数则使用 /dev/tty
```

下面介绍的特殊“变量”提供了对传递的参数的总数的访问，以及一次对所有参数的访问：

$#

提供传递到Shell脚本或函数的参数总数。当你是为了处理选项和参数而建立循环时，它会很有用（在稍后的6.4节里会说明）。举例如下：

```
while [ $# != 0 ]                    以shift逐渐减少$#，循环将会终止
do
    case $1 in
    ...                              处理第一个参数
    esac
    shift                            移开第一个参数（见稍后内文说明）
done
```

$*, $@

一次表示所有的命令行参数。这两个参数可用来把命令行参数传递给脚本或函数所执行的程序。

"$*"

将所有命令行参数视为单个字符串。等同于"$1 $2 ..."。$IFS的第一个字符用来作为分隔字符，以分隔不同的值来建立字符串。举例如下：

```
printf "The arguments were %s\n" "$*"
```

"$@"

将所有命令行参数视为单独的个体，也就是单独字符串。等同于"$1" "$2" ...。这是将参数传递给其他程序的最佳方式，因为它会保留所有内嵌在每个参数里的任何空白。举例如下：

```
lpr "$@"                             显示每一个文件
```

set命令可以做的事很多（详见7.9.1节说明）。调用此命令而未给予任何选项，则它会设置位置参数的值，并将之前存在的任何值丢弃：

```
set -- hi there how do you do        -- 会结束选项部分，自hi开始新的参数
```

shift命令是用来“截去(lops off)”来自列表的位置参数，由左开始。一旦执行shift，$1的初始值会永远消失，取而代之的是$2的旧值。$2的值，变成$3的旧值，以此类推。$#值则会逐次减1。shift也可使用一个可选的参数，也就是要位移的参数的计数。单纯的shift等同于shift 1。以下范例将这些操作串联在一起，并添加了注释：

```
$ set -- hello "hi there" greetings           设置新的位置参数
$ echo there are $# total arguments           显示计数值
there are 3 total arguments
$ for i in $*                                 循环处理每一个参数
> do   echo i is $i
> done
```

```
i is hello                              注意，内嵌的空白已消失
i is hi
i is there
i is greetings
$ for i in $@                           在没有双引号的情况下，$* 与 $@ 是一样的
> do   echo i is $i
> done
i is hello
i is hi
i is there
i is greetings
$ for i in "$*"                         加了双引号，$* 表示一个字符串
> do   echo i is $i
> done
i is hello hi there greetings
$ for i in "$@"                         加了双引号，$@ 保留真正的参数值
> do   echo i is $i
> done
i is hello
i is hi there
i is greetings
$ shift                                 截去第一个参数
$ echo there are now $# arguments       证明它已消失
there are now 2 arguments
$ for i in "$@"
> do   echo i is $i
> done
i is hi there
i is greetings
```

6.1.2.3 特殊变量

除了我们看过的特殊变量（例如 $# 及 $*）之外，Shell还有很多额外的内置变量。有一些也具有单一字符、非文字或数字字母的名称；其他则是全由大写字母组成的名称。

表6-3列出内置于Shell内的变量，以及影响其行为的变量。所有Bourne风格的Shell提供的变量都比这里所列的多很多，它们会影响交互模式下的使用，也可以在处理Shell程序时用于其他的用途。不过下面要说明的这些，是在写Shell程序时，可以完全倚赖实现可移植性脚本编程的变量。

表6-3：POSIX内置的Shell变量

变量	意义
#	目前进程的参数个数。
@	传递给当前进程的命令行参数。置于双引号内，会展开为个别的参数。
*	当前进程的命令行参数。置于双引号内，则展开为一单独参数。
-（连字号）	在引用时给予 Shell 的选项。

表 6-3：POSIX 内置的 Shell 变量（续）

变量	意义
?	前一命令的退出状态。
$	Shell 进程的进程编号（process ID）。
0（零）	Shell 程序的名称。
!	最近一个后台命令的进程编号。以此方式存储进程编号，可通过 wait 命令以供稍后使用。
ENV	一旦引用，则仅用于交互式 Shell 中；$ENV 的值是可展开的参数。结果应为要读取和在启动时要执行的一个文件的完整路径名称。这是一个 XSI 必需的变量。
HOME	根（登录）目录。
IFS	内部的字段分隔器；例如，作为单词分隔器的字符列表。一般设为空格、制表符（Tab），以及换行（newline）。
LANG	当前 locale 的默认名称；其他的 LC_* 变量会覆盖其值。
LC_ALL	当前 locale 的名称；会覆盖 LANG 与其他 LC_* 变量。
LC_COLLATE	用来排序字符的当前 locale 名称。
LC_CTYPE	在模式匹配期间，用来确定字符类别的当前 locale 的名称。
LC_MESSAGES	输出信息的当前语言的名称。
LINENO	刚执行过的行在脚本或函数内的行编号。
NLSPATH	在 $LC_MESSAGES(XSI)所给定的信息语言里，信息目录的位置。
PATH	命令的查找路径。
PPID	父进程的进程编号。
PS1	主要的命令提示字符串。默认为 "$ "。
PS2	行继续的提示字符串。默认为 "> "。
PS4	以 set -x 设置的执行跟踪的提示字符串。默认为 "+ "。
PWD	当前工作目录。

特殊变量 $$ 可在编写脚本时用来建立具有唯一性的文件名（多半是临时的），这是根据 Shell 的进程编号建立文件名。不过，系统里还有一个 mktemp 命令也能做同样的事，这些都会在第 10 章中探讨。

6.1.3 算术展开

Shell 的算术运算符与 C 语言里的差不多，优先级与顺序也相同。表 6-4 列出支持的算术运算符，优先级由最高排列至最低。虽有些是（或包含）特殊字符，不过它们不需以反

斜杠转义，因为它们都置于$((...))语法中。这一语法如同双引号功能，除了内嵌双引号无须转义，见7.7节。

表6-4：算术运算数

运算符	意义	顺序
++ --	增加及减少，可前置也可放在结尾	由左至右
+ - ! ~	一元（unary）的正号与负号；逻辑与位的（bitwise）取反	由右至左
* / %	乘法、除法，与余数	由左至右
+ -	加法与减法	由左至右
<< >>	向左位移、向右位移	由左至右
< <= > >=	比较	由左至右
== !=	相等与不等	由左至右
&	位的AND	由左至右
^	位的Exclusive OR	由左至右
|	位的OR	由左至右
&&	逻辑的AND（简捷方式）	由左至右
||	逻辑的OR（简捷方式）	由左至右
?:	条件表达式	由右至左
= += -= *= /= %= &= ^= <<= >>= |=	赋值运算符	由右至左

可利用圆括号将子表达式语句块括起来。就像在C里一样：关系运算符（<、<=、>、>=、==与!=）产生数字结果中，1表示为真，0表示假。

例如：$((3 > 2))的值为1；$(((3 > 2) || (4 <= 1)))也为1，因为这两个子表达式里至少有一个为真。

对逻辑的AND与OR运算符而言，任何的非0值函数都为真：

```
$ echo $((3 && 4))                3 与 4 都为“真”
1
```

非0值都为真的用法，可用于所有从C衍生而来的语言，例如C++、Java以及awk。

如果你对C、C++或Java已有所了解，那么应该也熟悉表6-4所列出的运算符。如果不熟悉，这里我们就来进行一些简单的说明。

常规运算符的赋值形式，对于较为传统的更新变量方式而言，是一种方便的缩写。举例

来说，在许多语言中，你可能会写 x = x + 2，以为 x 增加 2。但 += 运算符可以让你使用更简洁的写法：$((x += 2))，指的是为 x 增加 2，且将得到的结果存储到 x。

由于加1或减1的使用相当频繁，因此 ++ 与 -- 运算符提供缩写形式完成它们。也就是：++ 是增加1，而 -- 是减1。这些都属于单元运算符（unary operator），现在就来看看它们的作用：

```
$ i=5
$ echo $((i++)) $i
5 6
$ echo $((++i)) $i
7 7
```

发生什么情况呢？这两种情况，都是 i 的值加1。但是，运算符返回的值会根据它与变量的相对位置而定。后缀式（postfix）的运算符（运算符出现在变量之后），在结果产生后，将旧值返回给变量，再执行变量加1的操作。相对地，前缀式（prefix）中，运算数则是放在变量的前面，先将变量加1，再返回新值给变量。-- 的工作方式和 ++ 类似，只不过它的操作是将变量减1，而不是加1。

注意： ++ 与 -- 运算符是可选的：实际上，不必非支持它们，但 bash 与 ksh93 都支持此功能。

标准规范里，允许实现时可支持额外运算符。ksh93 的所有版本都支持 C 的逗点运算符，最近的版本还可以使用 ** 支持取幂功能；bash 也支持这两者。

标准规范中仅描述使用常数值的算术。当参数计算先完成时－例如 $i，算术计算程序就只看到常数值。实际上，所有支持 $((...)) 的 Shell，都可以让用户在提供变量名称时，无须前置 $ 符号。

根据 POSIX 规定，算术运算使用的是 C 的带有正负号的长整数。ksh93 支持浮点运算，不过如果你对程序的可移植性有所要求，建议不要依赖它们。

6.2 退出状态

每一条命令，不管是内置的、Shell 函数，还是外部的，当它退出时，都会返回一个小的整数值给引用它的程序，这就是大家所熟知的程序的退出状态（exit statu）。在 Shell 下执进程序时，有许多方式可取用程序的退出状态。

6.2.1 退出状态值

以惯例来说，退出状态为0表示“成功”，也就是，程序执行完成且未遭遇任何问题。其

他任何的退出状态都为失败（注1）（我们稍后将介绍如何使用退出状态）。内置变量?（以$?访问它）包括了Shell最近一次所执行的一个程序的退出状态。

例如，当你输入ls时，Shell找到ls并执行该程序。当ls结束时，Shell会恢复ls的退出状态。请见下面的例子：

```
$ ls -l /dev/null                                          ls一个存在的文件
crw-rw-rw-  1 root  root  1, 3 Aug 30 2001 /dev/null       ls的输出
$ echo $?                                                  显示退出状态
0                                                          退出状态为成功
$ ls foo                                                   现在ls一个不存在的文件
ls: foo: No such file or directory                         ls的错误信息
$ echo $?                                                  显示退出状态
1                                                          退出状态指出：失败
```

POSIX标准定义了退出状态及其含义，见表6-5。

表6-5：POSIX的结束状态

值	意义
0	命令成功地退出。
> 0	在重定向或单词展开期间（~、变量、命令、算术展开，以及单词切割）失败。
1-125	命令不成功地退出。特定的退出值的含义，是由各个单独的命令定义的。
126	命令找到了，但文件无法执行。
127	命令找不到。
> 128	命令因收到信号而死亡。

令人好奇的是，POSIX留下退出状态128未定义，仅要求它表示某种失败。因为只有低位（low-order）的8个位会返回给父进程，所以大于255的退出状态都会替换成该值除以256之后的余数。

你的Shell脚本可以使用exit命令传递一个退出值给它的调用者。只要将一个数字传递给它，作为第一个参数即可。脚本会立即退出，并且调用者会收到该数字且作为脚本的退出值：

```
exit 42        给最后一个问题返回答案
```

注1： C与C++的程序员请注意，这个部分与你所使用的程序完全相反，请花点时间适应。

exit

语法

```
exit [ exit-value ]
```

用途

目的是从Shell脚本返回一个退出状态给脚本的调用者。

主要选项

无

行为模式

如果没有提供，则以最后一个执行命令的退出状态作为默认的退出状态。如果这就是你要的，则最好明白地在Shell脚本里这么写：

```
exit $?
```

6.2.2 if-elif-else-fi语句

使用程序的退出状态，最简单的方式就是使用if语句。一般语法如下：

```
if pipeline
   [ pipeline ... ]
then
   statements-if-true-1
[ elif pipeline
   [ pipeline ... ]
then
   statements-if-true-2
... ]
[ else
   statements-if-all-else-fails ]
fi
```

（方括号表示的是可选的部分，并非逐字输入。）Shell的语法是松散地建立在Algol 68之上，而后者是V7 Shell的作者Steven Bourne相当推崇的。Algol 68最有名的地方是在于：使用以方括号作为开始与结束的关键字将语句组织起来，而不是使用Algol 60与Pascal所使用的begin与end定界符，也不是使用因C而普及化并且也常出现在其他可程序化的UNIX工具里使用的{和}。

以我们手边的例子来看，你应该大致猜得到它的工作方式：Shell执行第一组介于if与then之间的语句块。如果最后一条执行的语句成功地退出，它便执行*statements-if-true-1*，否则，如果有elif，它会尝试下一组语句块。如果最后一条语句成功地退出，则会执行*statements-if-true-2*。它会以这种方式继续，执行相对应的语句块，直到它碰到一个成功退出的命令为止。

如果if或elif语句里没有一个为真，并且else子句存在，它会执行*statements-if-all-else-fails*。否则，它什么事也不做。整个if...fi语句的退出状态，就是在then或else后面的最后一个被执行命令的退出状态。如果无任何命令执行，则退出状态为0。举例如下：

```
if grep pattern myfile > /dev/null
then
    ...         模式在这里
else
    ...         模式不在这里
fi
```

如果myfile含有模式pattern，则grep的退出状态为0。如果无任何的行匹配此模式，则退出状态的值为1，且如果发生一个错误，则会具有一个大于1的值。Shell会根据grep的退出状态，选择要执行哪一组语句块。

6.2.3 逻辑的NOT、AND与OR

有时，以否定状态表达测试操作会比较容易些："如果John不在家，则…"，在Shell下，这种情况的做法是：将惊叹号放在管道（pipeline）前：

```
if ! grep pattern myfile > /dev/null
then
    ...         模式不在这里
fi
```

POSIX在1992标准中引进这种标记方式。你可能会看到较旧的Shell脚本使用冒号（:）命令，其实并没有做任何事，它只是为了处理下面的情况：

```
if grep pattern myfile > /dev/null
then
    :          # 不做任何事
else
    ...         模式不在这里
fi
```

除了以!来测试事情的相反面之外，你也常会需要以AND与OR结构来测试多重子条件（如果John在家，且他不忙，则……）。当你以&&将两个命令分隔时，Shell会先执行第一个。如果它成功地退出，则Shell执行第二个。如果第二个命令也成功地退出，则整个语句块视为已经成功：

```
if grep pattern1 myfile && grep pattern2 myfile
then
    ...         myfile包含两种模式
fi
```

相对的，||运算符则是用来测试两种条件中是否有一个结果为真：

```
if grep pattern1 myfile || grep pattern2 myfile
then
    ...          一个或是另一个模式出现
fi
```

这两种都是快捷（short-circuit）运算符，即当判断出整个语句块的真伪时，Shell会立即停止执行命令。举例来说，在 *command1* && *command2*下，如果 *command1* 失败，则整个结果不可能为真，所以 *command2* 也不会被执行；以此类推，*command1* || *command2* 指的就是：如果 *command1* 成功，那么也没有理由执行 *command2*。

不要尝试过度“简练”而使用 && 和 || 取代 if 语句。我们不反对简短且简单的事情，如下：

```
$ who | grep tolstoy > /dev/null && echo tolstoy is logged on
tolstoy is logged on
```

上面的实际做法是：执行 who | grep ...，且如果成功，就显示信息。而我们曾见过有厂商提供Shell脚本，所使用的是这样的结构：

```
some_command && {
    one command
    a second command
    and a third command
}
```

花括号将所有命令语句块在一起，只有在 *some_command* 成功时它们才被执行。使用if可以让它更为简洁：

```
if some_command
then
    one command
    a second command
    and a third command
fi
```

6.2.4 test 命令

test 命令可以处理Shell脚本里的各类工作。它产生的不是一般输出，而是可使用的退出状态。test 接受各种不同的参数，可控制它要执行哪一种测试。

test 命令有另一种形式：[...]，这种用法的作用完全与 test 命令一样。因此，下面是测试两个字符串是否相等的两个语句：

```
if test "$str1" = "$str2"        if [ "$str1" = "$str2" ]
then                             then
    ...                              ...
fi                               fi
```

test, […]

语法

test [*expression*]

[[*expression*]]

用途

为了测试Shell脚本里的条件，通过退出状态返回其结果。要特别注意的是：这个命令的第二种形式，方括号根据字面意义逐字地输入，且必须与括起来的*expression*以空白隔开。

主要选项与表达式

见表6-6与内文。

行为模式

test用来测试文件属性、比较字符串及比较数字。

警告

POSIX风格的表达式只是在真实系统上是可用表达式的一个子集。需留意可移植性的问题。要了解更多信息，可参考14.3.2节。

除了极旧的UNIX系统外，test都已内置于Shell中。由于内置命令会比外部命令先被找到，所以，要写一个简单的测试程序并将其执行文件命名为test会有点麻烦。这种情况下你必须以./test引用这样的程序（假设它们在当前目录内）。

POSIX将test的参数描述为“表达式”，有一元表达式（unary）和二元的（binary）表达式。通常，一元的表达式由看似一个选项的部分（例如，-d用来测试文件是否为目录）与一个相对应的运算数组成，后者基本上（但不一定）是一个文件名。二元的表达式则有两个运算数与一个内嵌的运算符，以执行某种比较操作。再者，当只有一个参数时，test会检查它是否为null字符串。完整的列表参见表6-6。

表6-6：test表达式

运算符	如果……则为真
string	*string*不是null
-b *file*	*file*是块设备文件
-c *file*	*file*是字符设备文件
-d *file*	*file*是目录
-e *file*	*file*存在

表6-6：test表达式（续）

运算符	如果…则为真
-f *file*	*file*为一般文件
-g *file*	*file*有设置它的setgid位
-h *file*	*file*是一符号连接
-L *file*	*file*是一符号连接（等同于 -h）
-n *string*	*string*是非null
-p *file*	*file*是一命名的管道（FIFO文件）
-r *file*	*file*是可读的
-S *file*	*file*是socket
-s file	file不是空的
-t *n*	文件描述符*n*指向一终端
-u *file*	*file*有设置它的setuid位
-w *file*	*file*是可写入的
-x *file*	*file*是可执行的，或*file*是可被查找的目录
-z *string*	*string*为null
s1 = *s2*	字符串*s1*与*s2*相同
s1 != *s2*	字符串*s1*与*s2*不相同
n1 -eq n2	整数*n1*等于*n2*
n1 -ne n2	整数*n1*不等于n2
n1 -lt n2	*n1*小于*n2*
n1 -gt n2	*n1*大于*n2*
n1 -le n2	*n1*小于或等于*n2*
n1 -ge *n2*	*n1*大于或等于n2

也可以测试否定的结果，只需前置!字符即可。下面是测试运行的范例：

```
if [ -f "$file" ]
then
    echo $file is a regular file
elif [ -d "$file" ]
then
    echo $file is a directory
fi

if [ ! -x "$file" ]
then
    echo $file is NOT executable
fi
```

在XSI兼容的系统里，test版本是较为复杂的。它的表达式可以与-a（作逻辑的AND）与-o（作逻辑的OR）结合使用。-a的优先级高于-o，而=与!=优先级则高于其他的二元运算符。在这里，也可以使用圆括号将其语句括起来以改变计算顺序。

注意： 在使用-a和-o（它们是test运算符）与&&和||（它们是Shell运算符）之间是有一个差异点。

```
if [ -n "$str" -a -f "$file" ]        一个 test 命令，两种条件
if [ -n "$str" ] && [ -f "$file ]     两个命令，以快捷方式计算
if [ -n "$str" && -f "$file" ]        语法错误，见内文
```

第一个案例，test会计算两种条件。而第二个案例，Shell执行第一个test命令，且只有在第一个命令是成功的情况下，才会执行第二个命令。最后一个案例，&&为Shell运算符，所以它会终止第一个test命令，然后这个命令会抱怨它找不到结束的]字符，且以失败的值退出。即使test可以成功地退出，接下来的检查还是会失败，因为Shell（最有可能）找不到一个名为-f的命令。

---

ksh93与bash都支持一些额外的测试功能。在14.3.2节里有更多的相关信息。

POSIX的test算法介绍如表6-7所示。

表6-7：POSIX的test算法

参数	参数值	结果
0		退出状态为伪(1)
1	如果 $1 非 null	退出状态为真(0)
	如果 $1 为 null	退出状态为伪(1)
2	如果 $1 为 !	否定单一参数测试的结果，$2
	如果 $1 为一元运算符	运算符的测试结果
	其他情况	未定义
3	如果 $2 为二元运算符	运算符的测试结果
	如果 $1 为 !	否定双参数测试的结果，$2 $3
	如果 $1 是（且 $3 是）	单一参数测试的结果，$2(XSI)
	其他情况	未定义
4	如果 $1 为 !	否定三个参数测试的结果，$2 $3 $4
	如果 $1 是（且 $4 是）	两参数测试的结果，$2 $3(XSI)
	其他情况	未定义
> 4		未定义

为了可移植性，POSIX标准里建议对多重条件使用Shell层级测试，而非使用-a与-o运算符（我们也建议这么用）。举例如下：

```
if [ -f "$file" ] && ! [ -w "$file" ]
then
    # $file 存在且为一般文件，但不可写入
    echo $0: $file is not writable, giving up. >&2
    exit 1
fi
```

下面是几个使用test的诀窍：

需有参数

由于这个原因，所有的Shell变量展开都应该以引号括起来，这样test才能接受一个参数——即使它已变为null字符串。例如：

```
if [ -f "$file" ] ...        正确
if [ -f $file ] ...          不正确
```

在第二种情况下，万一$file恰巧是空的，则test接收到的参数会少于它所需要的，这将引发无法预料的奇怪行为。

字符串比较是很微妙的

特别是字符串值为空，或是开头带有一个减号时，test命令就会被混淆。因此有了一种比较难看不过广为使用的方式：在字符串值前置字母X（X的使用是随意的，但这是传统用法）。

```
if [ "X$answer" = "Xyes" ] ...
```

你会看到这种方式出现在许多Shell脚本中，事实上POSIX标准里的所有范例都是这么用。

将所有参数以引号括起来的算法仅适用于test，而这种算法在test的现代版本里是足够的，即使第一个参数的开头字符为减号也不会有问题。因此我们已经很少需要在新的程序里使用前置X的方式了。不过，如果可移植性最大化远比可读性重要，或许使用前置X的方式比较好（我们有时还是会这么做）。

test *是可以被愚弄的*

当我们要检查通过网络加载的文件系统访问时，就有可能将加载选项与文件权限相结合，以欺骗test，使其认为文件是可读取的，但事实是：操作系统根本就不让你访问这个文件。所以尽管：

```
test -r a_file && cat a_file
```

在理论上应该一定可行，但实际上会失败（注2）。针对这一点你可以做的就是加上一些其他层面的防御程序：

注2：Mike Haertel指出这种做法一直都不是百分百的可靠：a_file可能会在执行test与执行cat之间的处理期间修改。

```
if test -r a_file && cat a_file
then
    # cat worked, proceed on
else
    # attempt to recover, issue an error message, etc.
fi
```

只能作整数数字测试

你不能使用 test 做任何的浮点数算术运算。所有的数字测试只可处理整数（ksh93 认得浮点数字，但如果你在意可移植性的问题，最好就别用）。

例 6-1 给出了 2.6 节中 finduser 脚本的改良版。这个版本会测试 $#，即命令行参数编号，如果未提供，则显示错误信息。

例 6-1：用以寻找用户的脚本，需提供 username 参数

```
#! /bin/sh

# finduser --- 寻找是否有第一个参数所指定的用户登录

if [ $# -ne 1 ]
then
    echo Usage: finduser username >&2
    exit 1
fi

who | grep $1
```

6.3 case 语句

如果你需要通过多个数值来测试变量，可以将一系列 if 与 elif 测试搭配 test 一起使用：

```
if [ "X$1" = "X-f" ]
then
    ...         针对 -f 选项的程序代码
elif [ "X$1" = "X-d" ] || [ "X$1" = "X--directory" ]   # 允许长选项
then
    ...         针对 -d 选项的程序代码
else
    echo $1: unknown option >&2
    exit 1
fi
```

不过这么做的时候写起来很不顺手，也很难阅读（在 echo 命令里的 >&2，是传送输出到标准错误，这部分将在 7.3.2 节讨论）。相对地，Shell 的 case 结构应该用来进行模式匹配：

```
case $1 in
-f)
    ...               针对 -f 选项的程序代码
    ;;
-d | --directory)     # 允许长选项
    ...               针对 -d 选项的程序代码
    ;;
*)
    echo $1: unknown option >&2
    exit 1
    # 在“esac”之前的 ;; 形式是一个好习惯，不过并非必要
esac
```

这里我们看到，要测试的值出现在 case 与 in 之间。将值以双引号括起来虽然并非必要，但也无妨。要测试的值，根据 Shell 模式的列表依次测试，发现匹配的时候，便执行相对应的程序代码，直至 ;; 为止。可以使用多个模式，只要用 | 字符加以分隔即可，这种情况称为“or (或)”。模式里会包含任何的 Shell 通配字符，且变量、命令与算术替换会在它用作模式匹配之前在此值上被执行。

你可能会觉得在每个模式列表之后的不对称的右圆括号是有点奇怪；不过这也是 Shell 语言里不对称定界符的唯一实例 (instance)。(14.3.7 节里，我们将看到 bash 与 ksh 确实允许在模式列表前加上一个开头的“(”)。

最后的 * 模式是传统用法，但非必需的，它是作为一个默认的情况 (case)。这通常是在你要显示诊断信息并退出时使用。正如我们前面提及的，最后一个情况 (case) 不再需要结尾的 ;;，不过加上它，会是比较好的形式。

6.4 循环

除了 if 与 case 语句之外，还有 Shell 的循环结构也是非常好用的工具。

6.4.1 for 循环

for 循环用于重复整个对象列表，依次执行每一个独立对象的循环内容。对象可能是命令行参数、文件名或是任何可以以列表格式建立的东西。在 3.2.7.1 节里，我们曾提过这两行的脚本，用来更新一个 XML 文件：

```
mv atlga.xml atlga.xml.old
sed 's/Atlanta/&, the capital of the South/' < atlga.xml.old > atlga.xml
```

现在我们假定，比较可能出现的情况应该是拥有一些 XML 文件，再由这些 XML 文件集结成小册子。在此情况下，我们要做的应该是改变所有这些 XML 文件。所以 for 循环最适合这一情况：

```
for i in atlbrochure*.xml
do
    echo $i
    mv $i $i.old
    sed 's/Atlanta/&, the capital of the South/' < $i.old > $i
done
```

该循环将每个原始文件备份为副文件名为`.old`的文件，之后再使用`sed`处理文件以建立新文件。这个程序也显示文件名，作为执行进度的一种指示，这在有许多文件要处理时会有很大的帮助。

`for`循环里的`in`列表（list）是可选的，如果省略，Shell循环会遍历整个命令行参数。这就好像你已经输入了`for i in "$@"`：

```
for i        # 循环通过命令行参数
do
    case $i in
    -f) ...
        ;;
    ...
    esac
done
```

6.4.2 while与until循环

Shell的`while`与`until`循环，与传统程序语言的循环类似。语法为：

```
while condition              until condition
do                           do
    statements                   statements
done                         done
```

至于`if`语句，*condition*可以是简单的命令列表，或者是包含`&&`与`||`的命令。

`while`与`until`唯一的不同之处在于，如何对待*condition*的退出状态。只要*condition*是成功退出，`while`会继续循环。只要*condition*未成功结束，`until`则执行循环。例如：

```
pattern=...                      模式会控制字符串的缩简
while [ -n "$string" ]           当字符串不是空的时
do
    处理 $string 的当前值
    string=${string%$pattern}    截去部分字符串
done
```

实际上，`until`循环比`while`用得少，不过如果你在等待某个事件发生，它就很有用了。见例6-2。

例 6-2：使用 until，等待某个用户登录

```
# 等待特定用户登录，每 30 秒确认一次

printf "Enter username: "
read user
until who | grep "$user" > /dev/null
do
    sleep 30
done
```

你也可以将管道放入到 while 循环中，用来重复处理每一行的输入，如下所示：

```
产生数据  |
    while read name rank serial_no
    do
        ...
    done
```

以上述例子来说，while 循环的条件所使用的命令一直是 read。在稍后的 7.3.1 节中，我们会举一个比较实用的例子。在 7.6 节里，我们会告诉你，还可以使用管道将循环的输出传递给另一个命令。

6.4.3 break 与 continue

并非所有 Shell 里的东西都是直接来自 Algol 68。Shell 也从 C 借用了 break 与 continue 命令。这两个命令分别用来退出循环，或跳到循环体的其他地方。例 6-2 里 until ... do 的 wait-for-a-user（等待用户）脚本，可以用更传统的方式重写，见例 6-3：

例 6-3：使用 while 与 break，等待用户登录

```
# 等待特定用户登录，每 30 秒确认一次

printf "Enter username: "
read user
while true
do
    if who | grep "$user" > /dev/null
    then
        break
    fi

    sleep 30
done
```

true 命令什么事也不必做，只是成功地退出。这用于编写无限循环，即会永久执行的循环。在编写无限循环时，必须放置一个退出条件在循环体内，正如同这里所做的。另有一个 false 命令和它有点相似，只是较少用到，它也不做任何事，仅表示不成功的状态。false 命令常见于无限的 until false ... 循环中。

continue命令则用于提早开始下一段重复的循环操作，也就是在到达循环体的底部之前。

break与continue命令都接受可选的数值参数，可分别用来指出要中断（break）或继续多少个被包含的循环（如果循环计数需要的是一个在运行时可被计算的表达式时，可以使用 $((...)))。举例如下：

```
while condition1                    外部循环
do  ...
    while condition2                内部循环
    do  ...
        break 2                     外部循环的中断
    done
done
...                                 在中断之后，继续执行这里的程序
```

break与continue特别具备中断或继续多个循环层级的能力，从而以简洁的形式弥补了 Shell 语言里缺乏 goto 关键字的不足。

6.4.4 shift 与选项的处理

我们在6.1.2.2节中曾简短提及shift命令，它用来处理命令行参数的时候，一次向左位移一位（或更多位）。在执行 shift 之后，原来的 $1 就会消失，以 $2 的旧值取代，$2 的新值即为 $3 的旧值，以此类推，而 $# 的值也会逐次减少。shift 还接受一个可选的参数，也就是可以指定一次要移动几位：默认为 1。

通过结合 while、case、break 以及 shift，可以做些简单的选项处理，如下所示：

```
# 将标志变量设置为空值
file=   verbose=   quiet=   long=

while [ $# -gt 0 ]            执行循环直到没有参数为止
do
    case $1 in                检查第一个参数
    -f)    file=$2
           shift              移位退出“-f”，使得结尾的 shift 得到在 $2 里的值
           ;;
    -v)    verbose=true
           quiet=
           ;;
    -q)    quiet=true
           verbose=
           ;;
    -l)    long=true
           ;;
    --)    shift              传统上，以 -- 结束选项
           break
           ;;
```

```
        -*)     echo $0: $1: unrecognized option >&2
                ;;
        *)      break                  无选项参数，在循环中跳出
                ;;
        esac

        shift                          设置下一个重复
    done
```

在此循环结束之后，不同的标志变量都会设置，且可以使用test或case测试。任何剩下的无选项参数都仍然是可利用的，以便在 $@ 中做进一步的处理。

getopts命令简化了选项处理。它能理解POSIX选项中将多个选项字母组织到一起的用法，也可以用来遍历整个命令行参数，一次一个参数。

getopts 的第一个参数是列出合法选项字母的一个字符串。如果选项字母后面跟着冒号，则表示该选项需要一个参数，此参数是必须提供的。一旦遇到这样的选项，getopts会放置参数值到变量OPTARG中。另一个变量OPTIND包含下一个要处理的参数的索引值。Shell 会把该变量初始化为 1。

getopts的第二个参数为变量名称，在每次getopts调用时，该变量会被更新；它的值是找到的选项字母。当getopts找到不合法的选项时，它会将此变量设置为一个问号字符。我们以 getopts 重写前面的例子：

```
    # 设置标志变量为空
    file=   verbose=   quiet=   long=

    while getopts f:vql opt
    do
        case $opt in                   检查选项字母
        f)      file=$OPTARG
                ;;
        v)      verbose=true
                quiet=
                ;;
        q)      quiet=true
                verbose=
                ;;
        l)      long=true
                ;;
        esac
    done

    shift $((OPTIND - 1))          删除选项，留下参数
```

你会发现三个明显差异。首先，在 case 里的测试只是用在选项字母上，开头的减号被删除了。再者，针对 -- 的情况（case）也不见了：因为 getopts 已自动处理。最后也消失的就是针对不合法选项的默认情况：getopts 会自动显示错误信息。

getopts

语法

`getopts option_spec variable [ arguments ... ]`

用途

简化参数处理，并且让Shell脚本可以轻松地匹配于POSIX参数处理惯例。

主要选项

无

行为模式

当它被重复调用时（例如在while循环中），会依次通过给定的命令行参数，或者未提供则是"$@"，在--或第一个非选项参数处，或是碰到错误时，会以非零值退出。*option_spec*用来描述选项及它们的参数，见内文。

对每个合法的选项，设置variable为选项字母。如果选项有一个参数，则参数值会置于OPTARG里。在处理的结尾处，OPTIND会设置为第一个非选项参数的编号。见内文说明。

警告

ksh93版本的getopts会遵循POSIX，但还提供许多额外的功能。可参考ksh93文档，或是《Learning the Korn Shell》（O'Reilly）。

不过一般来说，在脚本里处理错误会比使用getopts的默认处理要容易。将冒号（:）置于选项字符串中作为第一个字符，可以使得getopts以两种方式改变它的行为：首先，它不会显示任何错误信息；第二，除了将变量设置为问号之外，OPTARG还包含了给定的不合法选项字母。以下便是选项处理循环的最后版本：

```
# 设置标志变量为空
file=   verbose=   quiet=   long=

# 开头的冒号，是我们处理错误的方式
while getopts :f:vql opt
do
    case $opt in              检查选项字母
    f)      file=$OPTARG
            ;;
    v)      verbose=true
            quiet=
            ;;
    q)      quiet=true
            verbose=
            ;;
    l)      long=true
            ;;
```

```
    '?')   echo "$0: invalid option -$OPTARG" >&2
           echo "Usage: $0 [-f file] [-vql] [files ...]" >&2
           exit 1
           ;;
    esac
done

shift $((OPTIND - 1))          删除选项，留下参数
```

警告： OPTIND变量是父脚本与其引用的任何函数所共享的。要使用getopts来解析自己的参数的函数，应将OPTIND重设为1。我们不建议在父脚本的选项处理循环中调用这样的函数（基于此，ksh93给每个函数它自己私有的OPTIND版本。再次提醒留意这点）。

6.5 函数

就像其他的程序语言一样，函数（function）是指一段单独的程序代码，用以执行一些定义完整的单项工作。在大型程序里，函数可以在程序的多个地方使用（调用）。

函数在使用之前必须先定义。这可通过在脚本的起始处，或是将它们放在另一个独立文件里且以点号（.）命令来取用（source）它们（.命令在稍后7.9节中会作说明）。定义方式如例6-4所示。

例6-4：等待用户登录 – 函数版

```
# wait_for_user --- 等待用户登录
#
# 语法 : wait_for_user user [ sleeptime ]

wait_for_user () {
    until who | grep "$1" > /dev/null
    do
        sleep ${2:-30}
    done
}
```

函数被引用（执行）的方式与命令相同：提供函数名称与任何相对应的参数。wait_for_user函数可以以两种方式被引用：

```
wait_for_user tolstoy           等待用户tolstoy，每30秒检查一次

wait_for_user tolstoy 60        等待用户tolstoy，每60秒检查一次
```

在函数体中，位置参数（$1、$2、…、$#、$*，以及$@）都是函数（function）的参数。父脚本的参数则临时地被函数参数所掩盖（shadowed）或隐藏。$0依旧是父脚本的名称。当函数完成时，原来的命令行参数会恢复。

在 Shell 函数里，return 命令的功能与工作方式都与 exit 相同：

```
answer_the_question () {
    ...
    return 42
}
```

需注意的是：在 Shell 函数体里使用 exit，会终止整个 Shell 脚本！

因为 return 语句会返回一个退出值给调用者，所以你可以在 if 与 while 语句里使用函数。举例来说，可使用 Shell 的函数架构取代 test 所执行的两个字符串的比较：

```
# equal --- 比较两个字符串

equal() {
    case "$1" in
    "$2")    return 0 ;;   # 两字符串匹配
    esac
    return 1               # 不匹配
}

if equal "$a" "$b" ...

if ! equal "$c" "$d" ...
```

return

语法

return [*exit-value*]

用途

返回由 Shell 函数得到的退出值给调用它的脚本。

主要选项

无

行为模式

如果未提供参数，则使用默认退出状态，也就是最后一个执行的命令的退出状态。如果这就是你要的，那么严谨的 Shell 函数写法为：

```
return $?
```

警告

有些 Shell 允许在脚本里使用 return，但如果用于函数体之外，则视为等同于 exit。这种用法并不建议，因为会出现可移植性的困扰。

有一个项目在这里需要注意：在 case 模式列表里使用双引号。这么做会强制该值视为字面上的字符串，而非 Shell 模式。不过在 $1 上使用引号则无伤大雅，但在这里没有必要。

函数也有像命令那样会返回整数的退出状态值：零表示成功，非零则为失败。如果要返回其他的值，函数应该设置一个全局性Shell变量，或是利用父脚本捕捉它（使用命令替换，见7.6节），显示其值。

```
myfunc () {
    ...
}
...
x=$(myfunc "$@")            调用myfunc，并存储输出
```

5.5节里的例5-6显示了一个含有9个步骤的管道，从输入文件中产生一个SGML/XML标签的排序列表。它仅在命令行所指定的一个文件上运作。我们现在可以使用`for`循环处理参数，并利用Shell函数封装管道，以利于处理多个文件。修改后的脚本见例6-5。

例6-5：从多个文件中，产生SGML标签列表

```
#! /bin/sh -
# 读取一个或多个在命令行上所提供的含有像<tag>word</tag>这样标记的
# HTML/SGML/XML文件，并将其以tab分隔列表内容为：
#      计数值  单词   标签    文件名
# 由小至大排序单词与标签，
# 将输出产生至标准输出上。
#
# 语法 :
#       taglist xml-files

process() {
  cat "$1" |
    sed -e 's#systemitem *role="url"#URL#g' -e 's#/systemitem#/URL#' |
      tr ' (){}[]' '\n\n\n\n\n\n\n' |
        egrep '>[^<>]+</' |
          awk -F'[<>]' -v FILE="$1" \
              '{ printf("%-31s\t%-15s\t%s\n", $3, $2, FILE) }' |
            sort |
              uniq -c |
                sort -k2 -k3 |
                  awk '{
                          print ($2 =  = Last) ? ($0 " <----") : $0
                          Last = $2
                       }'
}

for f in "$@"
do
    process "$f"
done
```

函数（至少在POSIX Shell里）没有提供局部变量（注3）。因此所有的函数都与父脚本

注3： bash、ksh88、ksh93以及zsh都提供局部变量功能，但语法不尽相同。

共享变量；即，你必须小心留意不要修改父脚本里不期望被修改的东西，例如PATH。不过这也表示其他状态是共享的，例如当前目录与捕捉信号（信号与捕捉将在13.3.2节讨论）。

6.6 小结

变量在正式一点的程序里是必备项目。Shell的变量会保存字符串值，而大量的运算符可在${*var*...}里使用，让你控制变量替换的结果。

Shell提供许多特殊变量（那些具有非文本和数字字符名称的，例如$?与$!），用来访问特殊信息，例如命令退出状态。Shell也有许多预先定义的特殊变量，例如PS1——用来设置主要提示字符串。位置参数与$*和$@这类的特殊变量，则用来在脚本（或函数）被引用时，让用户可以访问被使用的参数。env、export以及readonly则用来控制环境。

$((...))的算术展开提供完整的算术运算能力，且使用与C相同的运算符与优先级。

程序的退出状态是一个小的整数，可以在程序完成后，供引用者使用；Shell脚本使用exit命令来做这件事，而Shell函数则使用return命令。Shell脚本可以取得在特殊变量$?内执行的最后一个命令的退出状态。

退出状态可以搭配if、while与until语句来进行流程控制，也可与!、&&，以及||运算符搭配使用。

test命令及其别名[...]，可测试文件属性和字符串值与数值，在if、while以及until语句里，它也相当有用。

for提供遍历整组值的循环机制，这整组的值可以是字符串、文件名或其他等等。while与until提供比较传统的循环方式，加上break与continue提供额外的循环控制。case语句提供一个多重比较的功能，类似C与C++里的switch语句。

getopts、shift与$#提供处理命令行的工具。

最后，Shell函数可将相关命令组织到一起，之后再将它视为一个独立单元调用使用。它们有点像Shell脚本，只不过它将命令存放在内存里，这样会更有效率，且它们还能影响引用脚本的变量与状态（例如当前目录）。

第7章

输入/输出、文件与命令执行

本章将完整介绍Shell语言。首先讨论的是文件：如何以不同的方式处理输入/输出和产生文件名。接着是命令替换，也就是让你使用一个命令的输出作为命令行的参数。然后，我们继续将重点放在命令行上，讨论Shell提供的各类引用（quoting）。最后，则是深入探讨命令执行顺序，并针对内建于Shell里的命令作介绍。

7.1 标准输入、标准输出与标准错误输出

标准输入/输出（Standard I/O）可能是软件工具设计原则里最基本的观念了。它的构想是：程序应有一个数据来源、数据出口（数据要去哪），以及报告问题的地方。它们分别叫做标准输入（standard input）、标准输出（standard output）和标准错误输出（standard error）。程序应该不知道也不在意其输入与输出背后是哪种设备，这些设备可能是磁盘文件、终端、磁带机、网络连接，或者甚至是另一个执行中的程序！程序可以预期，在它启动的时候，这些标准位置都已打开，且已经准备好可以使用了。

有很多UNIX程序都遵循这一设计理念。默认情况下，它们会读取标准输入、写入标准输出，并将错误信息传递给标准错误输出。正如我们在第5章所见到的，这样的程序我们称它为过滤器（filter），因为它们“过滤”数据流，每一个都会在数据流上执行某种运算，再通过管道，将它传递给下一个。

7.2 使用read读取行

read命令是将信息传递给Shell程序的重要方式之一：

```
$ x=abc ; printf "x is now '%s'. Enter new value: " $x ; read x
x is now 'abc'. Enter new value: PDQ
$ echo $x
PDQ
```

read

语法

```
read [ -r ] variable ...
```

用途

将信息读入一个或多个Shell变量。

主要选项

-r

原始读取，不作任何处理。不将行结尾处的反斜杠解释为续行字符。

行为模式

自标准输入读取行（数据）后，通过Shell字段切割的功能（使用$IFS）进行切分。第一个单词赋值给第一个变量，第二个单词则赋值第二个变量，以此类推。如果单词多于变量，则所有剩下的单词，全赋值给最后一个变量。read一旦遇到文件结尾（end-of-file），会以失败值退出。

如果输入行以反斜杠结尾，则read会丢弃反斜杠与换行字符，然后继续读取下一行数据。如果你有使用-r选项，那么read便会以字面意义读取最后的反斜杠。

警告

当你将read应用在管道里时，许多Shell会在一个分开的进程内执行它。在这种情况下，任何以read所设置的变量，都不会保留它们在父Shell里的值。对管道中间内的循环，也是这样。

read可以一次读取所有的值到多个变量里。这种情况下，在$IFS里的字符会分隔输入行里的数据，使其成为各自独立的单词。例如：

```
printf "Enter name, rank, serial number: "
read name rank serno
```

最典型的用法是处理/etc/passwd文件。其标准格式为7个以冒号隔开的字段：用户名称、加密的密码、数值型用户ID、数值型组ID、全名、根目录与登录Shell。例如：

```
jones:*:32713:899:Adrian W. Jones/OSD211/555-0123:/home/jones:/bin/ksh
```

你可以使用简单的循环逐行处理/etc/passwd：

```
while IFS=: read user pass uid gid fullname homedir Shell
do
    ...         处理每个用户的行
done < /etc/passwd
```

这个循环并不是说“当IFS等于冒号时，便读取……”，而是通过IFS的设置，让read使用冒号作为字段分隔字符，而并不影响循环体里使用的IFS值，也就是说，它只改变read所继承环境内的IFS值。这一点在6.1.1节已作过说明。while循环则在6.4节里说明。

当遇到输入文件结尾时，read会以非0值退出。这个操作将终止while循环。

乍看之下可能会觉得将/etc/passwd的重定向放置于循环体的结尾有点奇怪，不过这是必需的，这样一来，read才会在每次循环的时候看到后续的行。如果循环写成这个样子：

```
# 重定向的不正确使用
while IFS=: read user pass uid gid fullname homedir Shell < /etc/passwd
do
    ...            处理每个用户的行
done
```

就永远不会终止了!每次循环时，Shell都会再打开/etc/passwd一次，且read只读取文件的第一行!

while read ... do ... done < *file*还有一种替代方式，即在管道里把cat和循环一起使用：

```
# 较容易读取，不过使用cat会损失一点效率：
cat /etc/passwd |
  while IFS=: read user pass uid gid fullname homedir Shell
  do
      ...            处理每个用户的行
  done
```

有一个常见的小技巧：任何命令都能用来将输入通过管道传送给read。当read用在循环中时，这一技巧格外有用。在3.2.7节里，我们曾展示过这个简单的脚本，用来复制整个目录树：

```
find /home/tolstoy -type d -print       |      寻找所有目录
  sed 's;/home/tolstoy/;/home/lt/;'     |      更改名称，留意使用的是分号定界符
    sed 's/^/mkdir /'                   |      插入mkdir命令
      sh -x                                    以Shell的跟踪模式执行
```

上面的例子，其实可以容易地完成，而且从Shell程序员的观点来看更加自然，也就是使用循环：

```
find /home/tolstoy -type d -print       |      寻找所有目录
  sed 's;/home/tolstoy/;/home/lt/;'     |      更改名称，留意使用的是分号定界符
    while read newdir                          读取新的目录名
    do
        mkdir $newdir                          建立目录
    done
```

(我们注意到这个脚本并不完美：特别是它无法保留原始目录的所有权与使用权限)

如果输入单词多于变量时，最后剩下的单词全部被指定给最后一个变量。理想的行为应转义这个法则：使用read搭配单一变量，将一整行的输入读取到该变量中。

很早以前，read默认的行为便是将输入行的结尾反斜杠看作续行（line continuation）的指示字符。这样的一行会使得read舍弃反斜杠与换行字符的结合，且继续读取下一个输入行：

```
$ printf "Enter name, rank, serial number: " ; read name rank serno
Enter name, rank, serial number: Jones \
> Major \
>123-45-6789
$ printf "Name: %s, Rank: %s, Serial number: %s\n" $name $rank $serno
Name: Jones, Rank: Major, Serial number: 123-45-6789
```

偶尔你还是会想要读取一整行的时候，而不管那一行包含了什么。-r选项可以实现此目的（-r选项是特定于POSIX的，许多Bourne Shell并不支持），当给定-r选项时，read不会将结尾的反斜杠视为特殊字符：

```
$ read -r name rank serno
tolstoy \                                      只提供两个字段
$ echo $name $rank $serno
tolstoy \                                      $serno是空的
```

7.3 关于重定向

我们已经介绍且使用过基本的输出入重定向运算符：<、>、>>，以及|。在本节，我们要看看还有哪些运算符可以使用，并介绍文件描述符（file-descriptor）处理的重要话题。

7.3.1 额外的重定向运算符

这里是Shell提供的额外运算符：

使用 set -C *搭配*

POSIX Shell提供了防止文件意外截断的选项：执行set -C命令可打开Shell所谓的禁止覆盖（noclobber）选项，当它在打开状态下时，单纯的>重定向遇到目标文件已存在时，就会失败。>| 运算符则可令noclobber选项失效。

提供行内输入（inline input）的<< 与 <<-

使用*program << delimiter*，可以在Shell脚本正文内提供输入数据。

这样的数据叫作嵌入文件（here document）。默认情况下，Shell可以在嵌入文件正文内做变量、命令和算术替换：

```
cd /home                    移到根目录的顶端
du -s *        |            产生原始磁盘用量
  sort -nr     |            以数字排序，最高的在第一个
    sed 10q    |            在前 10 行之后就停止
      while read amount name
      do
          mail -s "disk usage warning" $name << EOF
Greetings. You are one of the top 10 consumers of disk space
on the system.  Your home directory uses $amount disk blocks.

Please clean up unneeded files, as soon as possible.

Thanks,

Your friendly neighborhood system administrator.
EOF
      done
```

这个范例将电子邮件发送给系统上前十名的“磁盘贪婪户”，要求它们清理自己的根目录（以我们的经验来说，这种信息多半没有用，不过这么做会让系统管理人员觉得好过些）。

如果定界符以任何一种形式的引号括起来，Shell 便不会处理输入的内文：

```
$ i=5                                                   设置变量
$ cat << 'E'OF                                          定界符已被引用
> This is the value of i: $i                            尝试变量参照
> Here is a command substitution: $(echo hello, world)  命令替换
> EOF
This is the value of i: $i                              冗长式地显示文字
Here is a command substitution: $(echo hello, world)
```

嵌入文件重定向器的第二种形式有一个负号结尾。这种情况下，所有开头的制表符（Tab）在传递给程序作为输入之前，都从嵌入文件与结束定界符（closing delimiter）中删除（注意：只有开头的制表字符会被删除，开头的空格则不会删除）。这么做，让 Shell 脚本更易于阅读了，让我们来看看例 7-1 的通知程序修正版：

例 7-1：给磁盘贪婪户的一封信

```
cd /home                  移到 home 目录顶端
du -s *       |           产生原始磁盘使用量
  sort -nr    |           以数字排序，最高的在第一个
    sed 10q   |           找到前 10 行就停下来
      while read amount name
      do
        mail -s "disk usage warning" $name <<- EOF
          Greetings. You are one of the top 10 consumers
          of disk space on the system.  Your home directory
          uses $amount disk blocks.

          Please clean up unneeded files, as soon as possible.

          Thanks,
```

```
    Your friendly neighborhood system administrator.
    EOF
done
```

以 <> 打开一个文件作为输入与输出之用

使用 *program* <> *file*，可供读取与写入操作。默认是在标准输入上打开 *file*。

一般来说，< 以只读模式打开文件，而 > 以只写模式打开文件。<> 运算符则是以读取与写入两种模式打开给定的文件。这交由 *program* 确定并充分利用；实际上，使用这个操作符并不需要太多的支持。

警告： <> 最初是出现在最早的 V7 Bourne Shell 上，不过并没有形成文档，且经验告诉我们，在很多环境下，它的运行会有点问题。基于此，它并未被大家广泛了解或使用。虽然它已在 1992 年的 POSIX 标准中标准化，但很多系统里的 /bin/sh 并不支持它。因此如果你对程序可移植性的要求很高，最好避免使用。

对于使用 >| 我们也有相同的警告，此功能来自于 Korn Shell，并在 1992 年已标准化，不过至今仍有一些系统不支持。

7.3.2 文件描述符处理

在系统内部，UNIX 是以一个小的整数数字，称为文件描述符（file descriptors），表示每个进程的打开文件。数字由零开始，至多到系统定义的打开文件数目的限制。传统上，Shell 允许你直接处理至多 10 个打开文件：文件描述符从 0 至 9（POSIX 标准将是否可以处理大于 9 的文件描述符，保留给各实现自行定义。bash 可以，但 ksh 则否）。

文件描述符 0、1 与 2，各自对应到标准输入、标准输出以及标准错误输出。如前所述，每个程序都是从附加到终端的这些文件描述符开始（不管是真的终端还是虚拟终端，例如X window）。到当前为止，最常见的操作便是变更这三个文件描述符其中一个的位置，不过也可能处理其他的变动。首先来看的是：将程序的输出传送到一个文件，并将其错误信息传到另一个文件：

```
make 1> results 2> ERRS
```

上面的命令是将 make（注 1）的标准输出（文件描述符为 1）传给 results，并将标准错误输出（文件描述符为 2）传给 ERRS（make 不会知道这之间的差异：它不知道也不关心，也并未传送输出或错误信息到终端）。将错误信息捕捉在一个单独的文件里是一种很实用的做法，你之后可以使用分页程序查阅它们，或使用编辑器修正问题。否则，

注 1： make 程序用于控制原始文件重新编译为目标文件（object file）。不过它的用法相当多，要了解更多信息，可参考《Managing Projects with GNU make》（O'Reilly）。

大量输出的错误信息会快速地卷过屏幕画面，你会很难找到需要的信息。另一种不同的方式就更利落了，直接舍弃错误信息：

```
make 1> results 2> /dev/null
```

1> results 里的 1 其实没有必要，供输出重定向的默认文件描述符是标准输出：也就是文件描述符 1。下个例子会将输出与错误信息送给相同的文件：

```
make > results 2>&1
```

重定向 > results 让文件描述符 1（标准输出）作为文件 results，接下来的重定向 2>&1 有两部分。2> 重定向文件描述符 2，也就是标准错误输出。而 &1 是 Shell 的语法：无论文件描述符 1 在哪里。在本例中，文件描述符 1 是 results 文件，所以那里就是文件描述符 2 要附加的地方。需特别留意的一点是：在命令行上，这 4 个字符 2>&1 必须连在一起，中间不能有任何空格。

在此，顺序格外重要：Shell 处理重定向时，由左至右。来看看此例：

```
make 2>&1 > results
```

上述命令，Shell 会先传送标准错误信息到文件描述符 1，这是仍为终端，然后文件描述符 1（标准输出）被改为 results。更进一步，Shell 会在文件描述符重定向之前处理管道，使我们得以将标准输出与标准错误输出都传递到相同的管道：

```
make 2>&1 | ...
```

最后要介绍的是可用来改变 Shell 本身 I/O 设置的 exec 命令。使用时，如果只有 I/O 重定向而没有任何参数时，exec 会改变 Shell 的文件描述符：

```
exec 2> /tmp/$0.log              重定向 Shell 本身的标准错误输出
exec 3< /some/file               打开新文件描述符 3
...
read name rank serno <&3         从该文件读取
```

注意： 重定向 Shell 的标准错误输出的第一个示例行，应该只用于脚本中。交互式 Shell 会在标准错误输出上显示它们的提示号；所以如果你交互式地执行此命令，就看不到提示号了！如果你希望取消（undo）标准错误输出的重定向，可以先把它复制到一个新文件以存储文件描述符。例如：

```
exec 5>&2              把原来的标准错误输出保存到文件描述符 5 (fd5) 上
exec 2> /tmp/$0.log    重定向标准错误输出
...                    执行各种操作…
exec 2>&5              将原始文件复制到文件描述符 2
exec 5>&-              关闭文件描述符 5，因为不再需要了
```

exec

语法

exec [*program* [*arguments* ...]]

用途

以新的程序取代Shell，或改变Shell本身的I/O设置。

主要选项

无

行为

搭配参数－也就是使用指定的程序取代Shell，以传递参数给它。如果只使用I/O重定向，则会改变Shell本身的文件描述符。

搭配上参数，exec还能起到另一个作用，即在当前Shell下执行指定的程序。换句话说，就是Shell在其当前进程中启动新程序。例如，想使用Shell做选项处理但大部分工作仍要由一些其他程序来完成时，你可以用这个方式：

```
while [ $# -gt 1 ]                                  循环遍历参数
do
    case $1 in                                      处理选项
    -f)     # code for -f here
            ;;
    -q)     # code for -q here
            ;;
    ...
    *)      break ;;                                没有选项，中断循环
    esac

    shift                                           移到下一个参数
done

exec real-app -q "$qargs" -f "$fargs" "$@"          执行程序

echo real-app failed, get help! 1>&2                紧急信息
```

使用此法时，exec为单向操作。也就是说：控制权不可能会回到脚本。唯一的例外只有在新程序无法被调用时。在该情况下，我们会希望有“紧急”代码，可显示信息，再完成其他可能的清除工作。

7.4 printf的完整介绍

我们曾在2.5.4节中介绍过printf命令。本节我们将完整地介绍它。

printf命令的完整语法有两个部分：

```
Printf format-string [arguments...]
```

第一个部分为描述格式规格的字符串，它的最佳提供方式是放在引号内的字符串常数。第二个部分为参数列表，例如字符串或变量值的列表，该列表需与格式规格相对应。格式字符串结合要以字面意义输出的文本，它使用的规格是描述如何在printf命令行上格式化一连串的参数。一般字符都按照字面上的意义输出。转义序列会被解释（与echo相似），然后输出为相对应的字符。格式指示符（format specifier）是以%字符开头且由已定义的字母集之一作为结尾，用来控制接下来相对应参数的输出。printf的转义序列见表7-1。

printf

语法

```
printf format [ string ... ]
```

用途

为了从Shell脚本中产生输出。由于printf的行为是由POSIX标准所定义，因此使用printf的脚本比使用echo更具可移植性。

主要选项

无

行为

printf使用format字符串控制输出。字符串里的纯字符都如实打印。echo的转义序列会被解释。包括%与一个字母的格式指示符（Format specifier），用来指示相对应的参数字符串的格式化，详见内文介绍。

表7-1：printf的转义序列

序列	说明
\a	警告字符，通常为ASCII的BEL字符
\b	后退
\c	抑制（不显示）输出结果中任何结尾的换行字符注；而且，任何留在参数里的字符、任何接下来的参数以及任何留在格式字符串中的字符，都被忽略
\f	换页（formfeed）
\n	换行
\r	回车（Carriage return）
\t	水平制表符

表7-1：printf的转义序列（续）

序列	说明
\v	垂直制表符
\\	一个字面上的反斜杠字符
ddd	表示1到3位数八进制值的字符。仅在格式字符串中有效
\0*ddd*	表示1到3位的八进制值字符

注：只在%b格式指示符控制下的参数字符串中有效。

printf对转义序列的处理可能会让人觉得混淆。默认情况下，转义序列只在格式字符串中会被特别对待，也就是说，出现在参数字符串里的转义序列不会被解释：

```
$ printf "a string, no processing: <%s>\n" "A\nB"
a string, no processing: <A\nB>
```

当你使用%b格式指示符时，printf会解释参数字符串里的转义序列：

```
$ printf "a string, with processing: <%b>\n" "A\nB"
a string, with processing: <A
B>
```

无论是在格式字符串内还是在使用%b所打印的参数字符串里，大部分的转义序列都是被相同对待（如表7-1所示）。无论如何，\c与\0*ddd*只有搭配%b使用才有效，而*ddd*只有在格式字符串里才会被解释。

现在应大致可以作结论，格式指示符为printf提供了强大的功能和灵活性。格式规格字母请见表7-2。

表7-2：printf格式指示符

项目	说明
%b	相对应的参数被视为含有要被处理的转义序列之字符串。见表7-1
%c	ASCII字符。显示相对应参数的第一个字符
%d, %i	十进制整数
%e	浮点格式（[-]*d.precisione* [+-]*dd*）
%E	浮点格式（[-]*d.precisionE* [+-]dd）
%f	浮点格式（[-]*ddd.precision*）
%g	%e或%f转换，看哪一个较短，则删除结尾的零
%G	%E或%f转换，看哪一个较短，则删除结尾的零
%o	不带正负号的八进制值

表7-2：printf格式指示符（续）

项目	说明
%s	字符串
%u	不带正负号的十进制值
%x	不带正负号的十六进制值。使用a至f表示10至15
%X	不带正负号的十六进制值。使用A至F表示10至15
%%	字面意义的%

根据POSIX标准：浮点格式%e、%E、%f、%g与%G是"不需要被支持"。这是因为awk支持浮点算数运算，且有它自己的printf语句。这样，Shell程序中需要将浮点数值进行格式化的打印时，可使用小型awk程序实现。然而，内建于bash、ksh93和zsh中的printf命令都支持浮点格式。

printf命令可用来指定输出字段的宽度以及进行对齐操作。为实现此目的，接在%后面的格式表达式可采用三个可选用的修饰符(modifier)以及前置的格式指示符(format specifier)：

```
%flags width.precision format-specifier
```

输出字段的width为数字值。指定字段宽度时，字段的内容默认为向右对齐，如果你希望文字向左靠，必须指定-标志。这样："%-20s"会在一个有20个字符宽度的字段里，输出一个向左对齐的字符串。如果字符串少于20个字符，则字段将以空白填满。下面的例子里，|是输出，以表示字段的实际宽度。第一个例子为向右对齐文字：

```
$ printf "|%10s|\n" hello
|     hello|
```

下一个例子则为向左对齐文字：

```
$ printf "|%-10s|\n" hello
|hello     |
```

*precision*修饰符是可选用的。对十进制或浮点数值而言，它可以控制数字出现于结果中的位数。对字符串值而言，它控制将要打印的字符串的最大字符数。具体的含义会因格式指示符而有不同，见表7-3。

表7-3：精度的意义

转换	精度含义
%d, %i, %o, %u, %x, %X	要打印的最小位数。当值的位数少于此数字时，会在前面补零。默认精度（precision）为1

表7-3：精度的意义（续）

转换	精度含义
%e，%E	要打印的最小位数。当值的位数少于此数字时，会在小数点后面补零，默认精度为6。精度为0则表示不显示小数点
%f	小数点右边的位数
%g，%G	有效位数（significant digit）的最大数目
%s	要打印字符的最大数目

下面是几个精度的例子：

```
$ printf "%.5d\n" 15
00015
$ printf "%.10s\n" "a very long string"
a very lon
$ printf "%.2f\n" 123.4567
123.46
```

C函数库的printf()函数允许通过参数列表里的额外数值动态地指定宽度及精度。POSIX标准不支持此功能，相反地，它会建议你在格式字符串里使用Shell变量值（注2）。举例如下：

```
$ width=5  prec=6  myvar=42.123456
$ printf "|%${width}.${prec}G|\n" $myvar      POSIX
|42.1235|
$ printf "|%*.*G|\n" 5 6 $myvar              ksh93 与 bash
|42.1235|
```

最后要介绍的是：在字段宽度与精度前放置一个或多个标志的用法。我们已介绍过使用-标志可以让字符串向左对齐。完整的标志列表如表7-4所示。

表7-4：printf的标志

字符	意义
-	将字段里已格式化的值向左对齐。
空白（space）	在正值前置一个空格，在负值前置一个负号。
+	总是在数值之前放置一个正号或负号，即便是正值也是。
#	下列形式选择其一：%o有一个前置的0；%x与%X分别有前置的0x与0X。%e、%E与%f总是在结果中有一个小数点；%g与%G为没有结尾的零。
0	以零填补输出，而非空白。这仅发生在字段宽度大于转换后的情况下。在C语言里，该标志应用到所有输出格式，即使是非数字的值也一样。对于printf命令而言，它仅应用到数值格式。

注2：　某些printf版本，例如ksh93与bash下的版本，确实支持动态的宽度与精度指定。

再举例说明：

```
$ printf "|%-10s| |%10s|\n" hello world     字符串向左、向右对齐
|hello     | |     world|
$ printf "|% d| |% d|\n" 15 -15             空白标志
| 15| |-15|
$ printf "%+d %+d\n" 15 -15                 + 标志
+15 -15
$ printf "%x %#x\n" 15 15                   # 标志
f 0xf
$ printf "%05d\n" 15                        0 标志
00015
```

对于转换指示符 %b、%c 与 %s 而言，相对应的参数都视为字符串。否则，它们会被解释为 C 语言的数字常数（开头的 0 为八进制，以及开头的 0x 与 0X 为十六进制）。更进一步来说，如果参数的第一个字符为单引号或双引号，则相对应的数值是字符串的第二个字符的 ASCII 值：

```
$ printf "%s is %d\n" a "'a"
a is 97
```

当参数多于格式指示符时，格式指示符会根据需要再利用。这种做法在参数列表长度未知时是很方便的，例如来自通配符表达式。如果留在格式（format）字符串里剩下的指示符比参数多时，如果是数值转换，则遗漏的值会被看成是零，如果是字符串转换，则被视为空字符串（虽然可以这么用，但比较好的方式应该是确认你提供的参数数目，与格式字符串所预期的数目是一样的）。如果 printf 无法进行格式的转换，它便会返回一个非零的退出状态。

7.5 波浪号展开与通配符

Shell 有两种与文件名相关的展开。第一个是波浪号展开（tilde expansion），另一个则有很多种叫法，有人称之为通配符展开式（wildcard expansion），有人则称之为全局展开（globbing）或是路径展开（pathname expansion）。

7.5.1 波浪号展开

如果命令行字符串的第一个字符为波浪号（~），或者变量指定（例如 PATH 或 CDPATH 变量）的值里任何未被引号括起来的冒号之后的第一个字符为波浪号（~）时，Shell 便会执行波浪号展开。

波浪号展开的目的，是要将用户根目录的符号型表示方式，改为实际的目录路径。可以采用直接或间接的方式指定执行此程序的用户，如未明白指定，则为当前的用户：

```
$ vi ~/.profile                与vi $HOME/.profile相同
$ vi ~tolstoy/.profile         编辑用户tolstoy的.profile文件
```

以第一个例子来看，Shell将~换成$HOME，也就是当前用户的根目录。第二个例子，则是Shell在系统的密码数据库里，寻找用户tolstoy，再将~tolstoy置换为tolstoy的根目录。

注意： 波浪号展开最先是出现于Berkeley C Shell —— csh中。主要是交互式功能。事实证明它的确很受欢迎，而且之后还被Korn Shell、bash及绝大多数现代Bourne形式的Shell采纳成为自已的功能。因此POSIX标准里也说明了它的应用方式。

不过（总是有“不过”），有许多商用UNIX的Bourne Shell不支持它。因此，如果你对Shell脚本的可移植性有所要求，请不要在你的脚本里使用波浪号展开。

使用波浪展开有两个好处。第一，它是一种简洁的概念表示方式，让查阅Shell脚本的人更清楚脚本在做的事。第二，它可以避免在程序里把路径名称直接编码。先看这个脚本片段：

```
printf "Enter username: "          显示提示信息
read user                          读取指名的用户
vi /home/$user/.profile            编辑该用户的.profile文件
...
```

前面的程序假设所有用户的根目录都在/home之下。如果这有任何变动（例如，用户子目录根据部门存放在部门目录的子目录下），那么这个脚本就得重写。但如果使用波浪号展开，就能避免重写的情况：

```
printf "Enter username: "          显示提示信息
read user                          读取指名的用户
vi ~$user/.profile                 编辑该用户的.profile文件
...
```

这么一来，无论用户根目录在哪里，程序都可以正常运作。

许多Shell如ksh88、ksh93、bash与zsh，都提供额外的波浪号展开，详见14.3.7节。

7.5.2 使用通配符

寻找文件名里的特殊字符，也是Shell提供的服务之一。当它找到这类字符时，会将它

们视为要匹配的模式，也就是，一组文件的规格，它们的文件名称都匹配于某个给定的模式。Shell 会将命令行上的模式，置换为符合模式的一组排序过的文件名（注 3）。

如果接触过MS-DOS下的简单命令行环境，你可能会觉得*.*这样的通配符很熟悉，它指的是当前目录下的所有文件名。UNIX Shell的通配符也类似，只不过功能更强大。基本的通配符见表 7-5 所示。

表 7-5：基本的通配符

通配符	匹配
?	任何的单一字符。
*	任何的字符字符串。
[*set*]	任何在 *set* 里的字符。
[!*set*]	任何不在 *set* 里的字符。

?通配符匹配于任何的单一字符，所以如果你的目录里含有 whizprog.c、whizprog.log 与 whizprog.o 这三个文件，与表达式 **whizprog.?**匹配的为 whizprog.c 与 whizprog.o，但 whizprog.log 则不匹配。

星号（*）是一个功能强大而且广为使用的通配符；它匹配于任何字符组成的字符串。表达式**whizprog.***符合前面列出的所有三个文件；网页设计人员也可以使用 ***.html** 表达式匹配他们的输入文件。

注意：MS-DOS、MS-Windows 与 OpenVMS 用户要注意的是：UNIX 文件名中的点号（.）并没有任何特殊之处（除了文件名开头的点号表示隐藏文件外）；点号只是一个字符而已。例如：ls * 会列出在当前目录下的所有文件；你不需要像在其他系统上一样使用 *.*。

剩下的通配符就是 *set* 结构了。set 是一组字符列表（例如 abc）、一段内含的范围（例如 a-z），或者是这两者的结合。如果希望破折号（dash）也是列表的一部分，只要把它放在第一个或最后一个就可以了。表 7-6（假定在 ASCII 环境下）有更详尽的解释。

注 3：　由于目录里的文件未按照次序排列，因此Shell会排序每个通配字符展开后的结果。在部分系统上，会根据适合于系统的位置而排序，但各个机器底层的整理顺序各有不同。UNIX 传统主义者可能会用 export LC_ALL=C 设置他们习惯的行为模式。这在先前的 2.8 节已讨论过。

表7-6：使用set结构的通配符

表达式	匹配的单一字符
[abc]	a、b或c
[.,;]	句点、逗点，或分号
[-_]	破折号或下划线
[a-c]	a、b或c
[a-z]	任何一个小写字母
[!0-9]	任何一个非数字字符
[0-9!]	任何一个数字或惊叹号
[a-zA-Z]	任何一个小写或大写字母
[a-zA-Z0-9_-]	任何一个字母、任何一个数字、下划线或破折号

在原来的通配符范例中，**whizprog.[co]**与**whizprog.[a-z]**两者都匹配whizprog.c与whizprog.o，但whizprog.log则不匹配。

在左方括号之后的惊叹号用来“否定”一个set。例如**[!.;]**符合句点与分号以外的任何一个字符；**[!a-zA-Z]**符合任何一个非字母的字符。

范围表示法固然方便，但你不应该对包含在范围内的字符有太多的假设。比较安全的方式是：分别指定所有大写字母、小写字母、数字，或任意的子范围（例如**[f-q]**、**[2-6]**）。不要想在标点符号字符上指定范围，或是在混用字母大小写上使用，像**[a-Z]**与**[A-z]**这样的用法，都不保证一定能确切地匹配出包括所有想要的字母，而没有其他不想要的字符。更大的问题是在于：这样的范围在不同类型的计算机之间无法提供完全的可移植性。

另一个问题是：现行系统支持各种不同的系统语言环境（locale），用来描述本地字符集的工作方式。很多国家的默认locale字符集与纯粹ASCII的字符集是不同的。为解决这些问题，POSIX标准提出了方括号表达式（bracket expression），用来表示字母、数字、标点符号及其他类型的字符，并且具有可移植性，这部分我们在3.2.1.1节里已讨论过。在正则表达式下的方括号表达式里也出现相同的元素，它们可被用在兼容POSIX的Shell内的Shell通配符模式中，不过你仍应避免将其应用在需可移植的Shell脚本里。

习惯上，当执行通配符展开时，UNIX Shell会忽略文件名开头为一个点号的文件。像这样的“点号文件（dot files）”通常用做程序配置文件或启动文件。像是Shell的$HOME/.profile、ex/vi编辑器的$HOME/.exrc，以及bash与gdb使用的GNU readline程序库的$HOME/.inputrc。

要看到这类文件，需在模式前面明确地提供一个点号。例如：

```
echo .*                         显示隐藏文件
```

你可以使用 -a（显示全部）选项，让 ls 列出隐藏文件：

```
$ ls -la
total 4525
drwxr-xr-x   39 tolstoy   wheel        4096 Nov 19 14:44 .
drwxr-xr-x   17 root      root         1024 Aug 26 15:56 ..
-rw-------    1 tolstoy   wheel          32 Sep  9 17:14 .MCOP-random-seed
-rw-------    1 tolstoy   wheel         306 Nov 18 22:52 .Xauthority
-rw-r--r--    1 tolstoy   wheel         142 Sep 19  1995 .Xdefaults
-rw-r--r--    1 tolstoy   wheel         767 Nov 18 16:20 .article
-rw-r--r--    1 tolstoy   wheel         158 Feb 14  2002 .aumixrc
-rw-------    1 tolstoy   wheel       18828 Nov 19 11:35 .bash_history
...
```

注意：再强调一次，隐藏文件只是个习惯用法。在用户层面的软件上它是这样的，但核心程序（kernel）并不认为开头带有一个点号的文件与其他文件有不同。

7.6 命令替换

命令替换（command substitution）是指Shell执行命令并将命令替换部分替换为执行该命令后的结果。这听起来有点绕舌，不过实际上相当简单。

命令替换的形式有两种。第一种是使用反引号 —— 或称重音符号（`...`）的方式，将要执行的命令框起来：

```
for i in `cd /old/code/dir ; echo *.c`    产生 /old/code/dir 下的文件列表
do                                        循环处理
   diff -c /old/code/dir/$i $i | more     在分页程序下比较旧版与新版的异同
done
```

这个Shell一开始执行 cd /old/code/dir ; echo *.c，产生的输出结果（文件列表）接着会成为 for 循环里所使用的列表。

反引号形式长久以来一直是供命令替换使用的方法，而且POSIX也支持它，因此许多已存在的Shell脚本都使用它。不过，所有最简单的用法很快会变成复杂的，特别是内嵌的命令替换及使用双引号时，都需要小心地转义反斜杠字符：

```
$ echo outer `echo inner1 \`echo inner2\` inner1` outer
outer inner1 inner2 inner1 outer
```

这个例子举得有点牵强，不过正说明了必须使用反引号的原因。命令执行顺序如下：

1. 执行echo inner2，其输出（为单词inner2）会放置到下一个要被执行的命令中。
2. 执行echo inner1 inner2 inner1，其输出（单词inner1 inner2 inner1）会放置到下一个要执行的命令中。
3. 最后，执行echo outer inner1 inner2 inner1 outer。

使用双引号，情况更糟：

```
$ echo "outer +`echo inner -\`echo \"nested quote\" here\`- inner`+ outer"
outer +inner -nested quote here- inner+ outer
```

为了更清楚明白，我们使用负号括住内部的命令替换，而使用正号框住外部的命令替换。简而言之，就是更为混乱。

由于使用嵌套的命令替换，有或没有引号，很快地就变得很难阅读，所以POSIX Shell采用Korn Shell里的一个功能。不用反引号的用法，改为将命令括在$(...)里。因为这种架构使用不同的开始定界符与结束定界符，所以看起来容易多了。以先前的例子来看，使用新语法重写如下：

```
$ echo outer $(echo inner1 $(echo inner2) inner1) outer
outer inner1 inner2 inner1 outer
$ echo "outer +$(echo inner -$(echo "nested quote" here)- inner)+ outer"
outer +inner -nested quote here- inner+ outer
```

这样是不是好看多了？不过要特别留意的是：内嵌的双引号不再需要转义。这种风格已广泛建议用在新的开发上，本书中有许多范例也是使用这种方法。

这边要看的是先前介绍过，使用for循环比较不同的两个目录下的文件版本，以新语法重写如下：

```
for i in $(cd /old/code/dir ; echo *.c)     产生/old/code/dir下的文件列表
do                                          循环处理
    diff -c /old/code/dir/$i $i             旧版本与新版本相比较
done | more                                 将所有结果经过分页程序
```

这里不同之处在于使用$(...)命令替换，以及将"整个"循环的输出，通过管道（pipe）送到more屏幕分页程序。

7.6.1 为head命令使用sed

之前在第3章的例3-1介绍过使用sed的head命令来显示文件的前*n*行。真实的head命令可加上选项，以指定要显示多少行，例如head -n 10 /etc/passwd。传统的POSIX

版本之前的head可指明行数作为选项（例如head -10 /etc/passwd），且许多UNIX的长期用户也习惯于使用此法执行head。

使用命令替换与sed，我们对Shell脚本稍作修改，使其与原始head版本的工作方式相同。见例 7-2。

例 7-2：使用 sed 的 head 命令的脚本，修订版

```
# head --- 打印前 n 行
#
# 语法 :   head -N file

count=$(echo $1 | sed 's/^-//')     # 截去前置的负号
shift                               # 移出 $1
sed ${count}q "$@"
```

当我们以head -10 foo.xml调用这个脚本时，sed最终是以sed 10q foo.xml被引用。

7.6.2 创建邮件列表

不同UNIX Shell的新版本不断地出现，而且分散在各站点的用户可以从/etc/Shells所列的Shell中选择自己的登录Shell。这样，如果以email通知用户有新版本的Shell更新且已安装的功能，这对系统管理而言会是很不错的事。

为了实现这个目的，我们必须先以登录Shell来识别用户，且产生邮件列表供安装程序用来公告新Shell版本。由于每封通知信息内容都不尽相同，我们也不是要建立一个直接传送邮件的脚本，只是要建立一个可用来寄送邮件的地址列表。邮件列表格式会因邮件用户端程序而有所不同，所以我们可以做一个合理的假设：最后完成的，只是一个以逗点分隔电子邮件地址的列表，一行一个或多个地址，而最后一个地址之后是否有逗点则不重要。

在这种情况下，较合理的方式应该是通过密码文件处理，为每个登录Shell建立一个输出文件，文件中每一行的用户名称都以逗点结束。这里是我们曾于第5章使用过的密码文件：

```
jones:*:32713:899:Adrian W. Jones/OSD211/555-0123:/home/jones:/bin/ksh
dorothy:*:123:30:Dorothy Gale/KNS321/555-0044:/home/dorothy:/bin/bash
toto:*:1027:18:Toto Gale/KNS322/555-0045:/home/toto:/bin/tcsh
ben:*:301:10:Ben Franklin/OSD212/555-0022:/home/ben:/bin/bash
jhancock:*:1457:57:John Hancock/SIG435/555-0099:/home/jhancock:/bin/bash
betsy:*:110:20:Betsy Ross/BMD17/555-0033:/home/betsy:/bin/ksh
tj:*:60:33:Thomas Jefferson/BMD19/555-0095:/home/tj:/bin/bash
george:*:692:42:George Washington/BST999/555-0001:/home/george:/bin/tcsh
```

脚本本身结合了变量与命令替换、read命令及while循环，整个脚本执行的代码不到10行!见例7-3。

例7-3：将密码文件转换为Shell邮寄列表

```
#! /bin/sh

# passwd-to-mailing-list
#
# 产生使用特定 Shell 的所有用户邮寄列表
#
# 语法 :
#   passwd-to-mailing-list < /etc/passwd
#   ypcat passwd | passwd-to-mailing-list
#   niscat passwd.org_dir | passwd-to-mailing-list

# 此操作或许有些过度小心：
rm -f /tmp/*.mailing-list

# 从标准输入读取：
while IFS=: read user passwd uid gid name home Shell
do
    Shell=${Shell:-/bin/sh} # 空的 Shell 字段意指 /bin/sh
    file="/tmp/$(echo $Shell | sed -e 's;^/;;' -e 's;/;-;g').mailing-list"
    echo $user, >> $file
done
```

每次读取密码文件的记录时，程序都会根据Shell的文件名产生文件名。sed命令会删除前置/字符，并将后续的每个/改成连字号。这段脚本会建立/tmp/bin-bash.mailing-list.这样形式的文件名。每个用户的名称与结尾的逗点都通过>>附加到特定的文件中。执行这个脚本后，会得到以下结果：

```
$ cat /tmp/bin-bash.mailing-list
dorothy,
ben,
jhancock,
tj,
$ cat /tmp/bin-tcsh.mailing-list
toto,
george,
$ cat /tmp/bin-ksh.mailing-list
jones,
betsy,
```

我们可以让这个建立邮件列表的程序更广泛地应用。例如，如果系统的进程统计(process accounting)是打开的，要为系统里的每个程序做一份邮件列表就很容易了，只要从进程统计记录中取出程序名称和执行过程序的用户姓名即可。注意，访问统计文件必须拥有root权限。每个厂商提供的统计软件都不一样，不过它们累积的数据的种类都差不多，所以只要稍作微调即可。GNU的统计摘要工具：sa（见*sa*(8)手册页）可产生类似下面这样的报告：

```
# sa -u
...
jones      0.01 cpu      377k mem      0 io gcc
...
```

也就是说，我们有一个以空白隔开的字段，第一个字段为用户名称而且最后一个字段为程序名称。这让我们可以简单地过滤该输出，使其看起来像是密码文件数据类型，再将其通过管道传递给我们的邮件列表程序来处理：

```
sa -u | awk '{ print $1 "::::::" $8 }' | sort -u | passwd-to-mailing-list
```

（sort 命令是用来排序数据；-u 选项则删除重复的行。）这个 UNIX 过滤程序与管道以及简单的数据标记，最漂亮的地方就在于简洁。我们不必编写新的邮件列表产生程序来处理统计数据：只需一个简单的 awk 步骤以及一个 sort，就可以让数据看起来就像我们可以处理的那样！

7.6.3 简易数学：expr

expr命令是UNIX命令里少数几个设计的不是那么严谨又很难用的一个。虽经过POSIX标准化，但我们非常不希望你在新程序里使用它，因为还有更多其他程序与工具做得比它更好。在Shell脚本的编写上，expr主要用于Shell的算术运算，所以我们把重点放这就好。如果你真有那么强的求知欲，可以参考 *expr*(1)的手册页了解更详尽的使用方法。

expr 的语法很麻烦：运算数与运算符必须是单个的命令行参数；因此我们建议你在这里大量使用空格间隔它们。很多 expr 的运算符同时也是 Shell 的 meta 字符，所以必须谨慎使用引号。

expr 被设置用在命令替换之内。这样，它会通过打印的方式把值返回到标准输出，而并非通过使用退出码（也就是 Shell 内的 $?）。

表 7-7 列出 expr 中优先级由小至大的运算符。我们将优先级相同的运算符组在一起。

表 7-7：expr 运算符

表达式	意义
e1 | *e2*	如果 *e1* 是非零值或非 null，则使用它的值。否则如果 *e2* 是非零值或非 null，则使用它的值。如果两者都不是，则最后值为零。
e1 & *e2*	如果 *e1* 与 *e2* 都非零值或非 null，则返回 *e1* 的值。否则，最后值为零。
e1 = *e2*	等于。
e1 != *e2*	不等于。
e1 < *e2*	小于。

表7-7：expr运算符（续）

表达式	意义
e1 <= *e2*	小于或等于。
e1 > *e2*	大于。
e1 >= *e2*	大于或等于。
	这些运算符，如果指示的比较为真，则会使得 expr 显示 1，否则显示 0。如果两个运算数都为整数，则以数字方式比较；如果不是，则以字符串方式比较。
e1 + *e2*	*e1* 与 *e2* 的加总。
e1 - *e2*	*e1* 与 *e2* 的相差。
e1 * *e2*	*e1* 与 *e2* 的相乘结果。
e1 / *e2*	*e1* 除以 *e2* 后的整数结果（截断）。
e1 % *e2*	*e1* 除以 *e2* 后的余数（截断）。
e1 : *e2*	*e1* 与 *e2* 的 BRE 匹配；详见 *expr*(1)的手册页。
(*expression*)	表达式 *expression* 的值；用于分组，大部分程序语言里都看得到。
integer	一个只包含数字的数目，允许前置负号，但却不支持一元的正号。
string	字符串值，不允许被误用为数字或运算符。

在新的代码里，你可以使用 test 或 $((...)) 进行这里的所有运算。正则表达式的匹配与提取，也可搭配 sed 或是 Shell 的 case 语句来完成。

这里有一个简单算术运算的例子。在真实的脚本里，循环体会做一些较有意义的操作，而不只是把循环变量的值显示出来：

```
$ i=1                                  初始化计数器
$ while [ "$i" -le 5 ]                 循环测试
> do
>   echo i is $i                       循环体：真正的代码在此
>   i=`expr $i + 1`                    循环计数器增值
> done
i is 1
i is 2
i is 3
i is 4
i is 5
$ echo $i                              显示最后结果
6
```

这类的算术运算，已经给出了你可能遇到的 expr 的使用方式的 99%。我们故意在这里使用 test（别名用法为 [...]）以及反引号的命令替换，因为这是 expr 的传统用法。在新的代码里，使用 Shell 的内建算术替换应该会更好：

```
$ i=1                              初始化计数器
$ while [ "$i" -le 5 ]             循环测试
> do
>   echo i is $i                   循环体：真正的代码在此
>   i=$((i + 1))                   循环计数器增值
> done
i is 1
i is 2
i is 3
i is 4
i is 5
$ echo $i                          显示最后的值
6
```

无论 expr 的价值如何，它支持 32 位的算术运算，也支持 64 位的算术运算 —— 在很多系统上都可以，因此，几乎不会有计数器溢出（overflow）的问题。

7.7 引用

引用（quoting）是用来防止 Shell 将某些你想要的东西解释成不同的意义。举例来说，如果你要命令接受含有 meta 字符的参数，如 * 或 ?，就必须将这些 meta 字符用引号引用起来。或更典型的情况是：你希望将某些可能被 Shell 视为个别参数的东西保持为单个参数，这时你就必须将其引用。这里是三种引用的方式：

反斜杠转义

字符前置反斜杠（\），用来告知 Shell 该字符即为其字面上的意义。这是引用单一字符最简单的方式：

```
$ echo here is a real star: \* and a real question mark: \?
here is a real star: * and a real question mark: ?
```

单引号

单引号（'...'）强制 Shell 将一对引号之间的所有字符都看作其字面上的意义。Shell 脚本会删除这两个引号，只单独留下被括起来的完整文字内容：

```
$ echo 'here are some metacharacters: * ? [abc] ` $ \'
here are some metacharacters: * ? [abc] ` $ \
```

不可以在一个单引号引用的字符串里再内嵌一个单引号。即便是反斜杠，在单引号里也没有特殊意义（某些系统里，像 echo 'A\tB' 这样的命令看起来像是 Shell 特别地处理反斜杠，其实不然，这是 echo 命令本身有特殊的处理方式，详见表 2-2）。

如需混用单引号与双引号，你可以小心地使用反斜杠转义以及不同引用字符串的连接来做到：

```
$ echo 'He said, "How'\''s tricks?"'
He said, "How's tricks?"
```

```
$ echo "She replied, \"Movin' along\""
She replied, "Movin' along"
```

不管你怎么处理，这种结合方式永远是很难读阅读的。

双引号

双引号（"..."）就像单引号那样，将括起来的文字视为单一字符串。只不过，双引号会确切地处理括起来文字中的转义字符和变量、算术、命令替换：

```
$ x="I am x"
$ echo "\$x is \"$x\". Here is some output: '$(echo Hello World)'"
$x is "I am x". Here is some output: 'Hello World'
```

在双引号里，字符$、"、` 与\，如需用到字面上的意义，都必须前置\。任何其他字符前面的反斜杠是不带有特殊意义的。序列\-*newline*会完全地被删除，就好像是用在脚本的正文中一样。

请注意，如范例所示，单引号被括在双引号里时就无特殊意义了，它们不必成对，也无须转义。

一般来说，使用单引号的时机是你希望完全不处理的地方。否则，当你希望将多个单词视为单一字符串，但又需要Shell为你做些事情的时候，请使用双引号，例如，将一个变量值与另一个变量值连接在一起，你就可以这么用：

```
oldvar="$oldvar $newvar"            将newvar的值附加到oldvar变量
```

7.8 执行顺序与eval

我们曾提到过的各类展开与替换都以定义好的次序完成。POSIX标准更提供了很多琐碎的细节。在这里，我们站在Shell程序设计人员的层面来看这些必须了解的东西。这里的解释，省略了许多小细节：例如复合命令的中间与结尾、特殊字符等。

Shell从标准输入或脚本中读取的每一行称为管道（pipeline）；它包含了一个或多个命令（command），这些命令被零或多个管道字符（|）隔开。事实上还有很多特殊符号可用来分隔单个的命令：分号（;）、管道（|）、&、逻辑AND（&&），还有逻辑OR（||）。对于每一个读取的管道，Shell都会将命令分割，为管道设置I/O，并且对每一个命令依次执行下面操作：

1. 将命令分割成*token*，是以固定的一组meta字符分隔，有空格、制表字符、换行字符、;、(、)、<、>、|与&。token的种类包括单词（word）、关键字（keyword）、输出入重定向器，以及分号。

   这是微妙的，但是变量、命令还有算术替换，都可以在Shell执行token认定的时候被执行。这就是为什么先前在7.5.1节所举的`vi ~$user/.profile`例子可以像预期的那样工作。

2. 检查每个命令的第一个 token，看看是否它是不带有引号或反斜杠的关键字 (keyword)。如果它是一个开放的关键字（if 与其他控制结构的开始符号，如{或()，则这个命令其实是一个复合命令（compound command）。Shell 为复合命令进行内部的设置，读取下一条命令，并再次启动进程。如果关键字非复合命令的开始符号（例如，它是控制结构的中间部分，像 then、else 或 do，或是结尾部分，例如 fi、done 或逻辑运算符），则 Shell 会发出语法错误的信号。

3. 将每个命令的第一个单词与别名（alia）列表对照检查。如果匹配，它便代替别名的定义，并回到步骤 1；否则，进行步骤 4（别名是给交互式 Shell 使用，因此我们在这里不谈）。回到步骤 1，允许让关键字的别名被定义：例如 alias aslongas=while or alias procedure=function。注意，Shell 不会执行递归（recursive）的别名展开：反而当别名展开为相同的命令时它会知道，并停止潜在的递归操作。可以通过引用要被保护的单词的任何部分而禁止别名展开。

4. 如果波浪号（~）字符出现在单词的开头处，则将波浪号替换成用户的根目录 ($HOME)。将 ~*user* 替换成 user 的根目录。

   波浪号替换（在支持此功能的 Shell 里）会发生在下面的位置：

   - 在命令行里，作为单词的第一个未引用字符
   - 在变量赋值中的 = 之后以及变量赋值中的任何:之后
   - 形式 ${*variable op word*}的变量替换里的 *word* 部分

5. 将任何开头为 $ 符号的表达式，执行参数（变量）替换。

6. 将任何形式为 $(*string*)或 \`*string*\` 的表达式，执行命令替换。

7. 执行形式 $((*string*))的算术表达式（arithmetic expression）。

8. 从参数、命令与算术替换中取出结果行的部分，再一次将它们切分为单词。这次它使用 $IFS 里的字符作为定界符，而不是使用步骤 1 的那组 meta 字符。

   通常，在 IFS 里连续多个重复的输入字符是作为单一定界符，这是你所期待的。这只有对空白字符（例如空格与制表字符）而言是真的。对于非空白字符，则不是这样的。举例来说，当读取以冒号分隔字段的 /etc/passwd 文件时，两个连续冒号所界定的是一个空字段：

```
while IFS=: read name passwd uid gid fullname homedir Shell
do
    ...
done < /etc/passwd
```

9. 对于 *、?，以及一对[...]的任何出现次数，都执行文件名生成（filename generation）的操作，也就是通配符展开。

10. 使用第一个单词作为一个命令，遵循7.9节中所述的查找次序；也就是，先作为一个特殊的内建命令，接着是作为函数，然后作为一般的内建命令，以及最后作为查找 $PATH 找到的第一个文件。

11. 在完成I/O重定向与其他同类型事项之后，执行命令。

如图7-1，引用可用来避开执行程序的不同部分。eval 命令可让你再经过一次这一流程。执行两次命令行的处理，看来似乎有点怪，不过这却是相当好用的功能：它让你编写一个脚本，可在任务中建立命令字符串，再将它们传递给 Shell 执行。这么一来，你可以让脚本聪明地修改它们自己执行时的行为（这在下一节会讨论）。

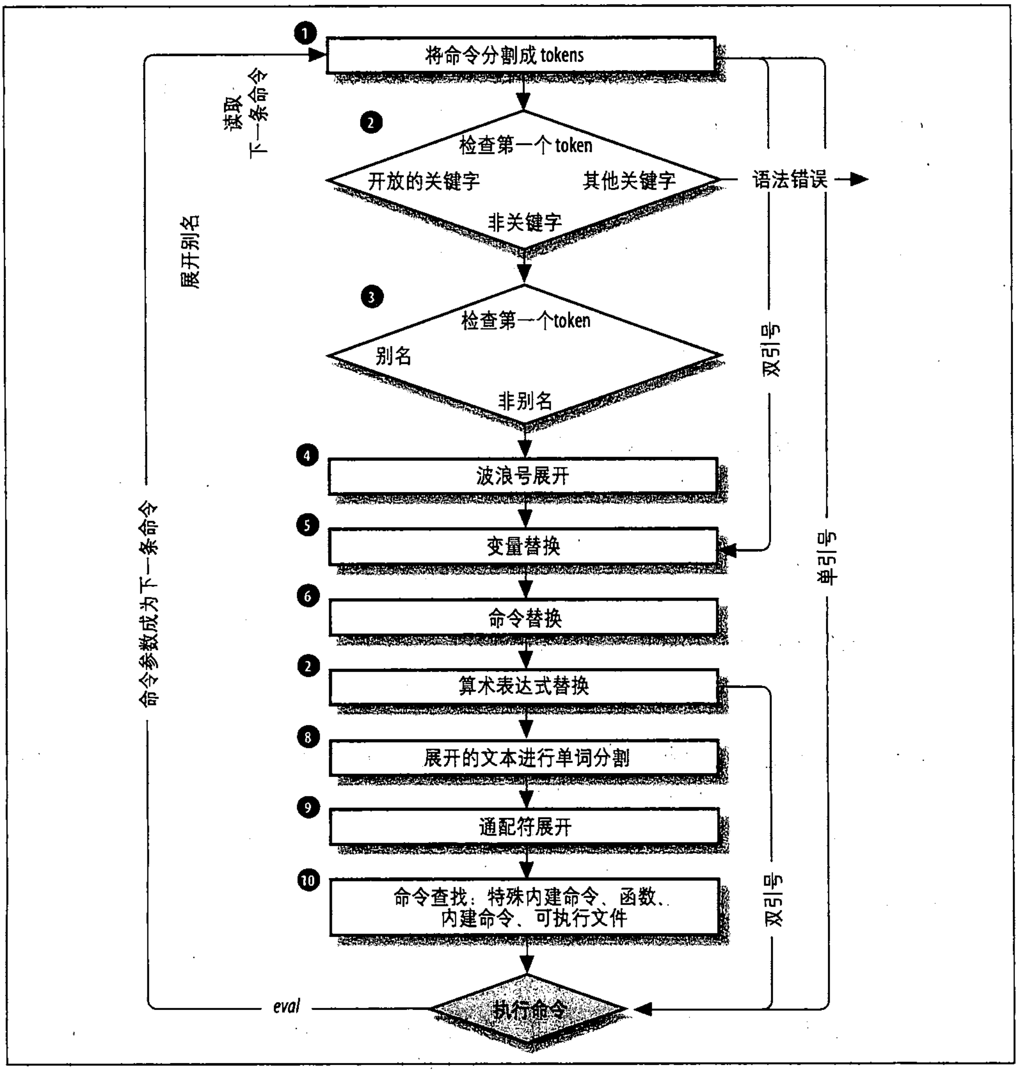


图7-1：命令行处理中的步骤

整个步骤顺序如图 7-1 所示，看起来有点复杂。当命令行被处理时，每一个步骤都是在 Shell 的内存里发生的；Shell 不会真地把每个步骤的发生显示给你看。所以，你可以假想这是我们偷窥 Shell 内存里的情况，从而知道每个阶段的命令行是如何被转换的。我们从这个例子开始说：

```
$ mkdir /tmp/x                                                建立临时性目录
$ cd /tmp/x                                                   切换到该目录
$ touch f1 f2                                                 建立文件
$ f=f y="a b"                                                 赋值两个变量
$ echo ~+/${f}[12] $y $(echo cmd subst) $((3 + 2)) > out     忙碌的命令
```

上述的执行步骤概要如下：

1. 命令一开始会根据 Shell 语法而分割为 token。最重要的一点是：I/O 重定向 >out 在这里是被识别的，并存储供稍后使用。流程继续处理下面这行，其中每个 token 的范围显示于命令下方的行上：

   ```
   echo ~+/${f}[12] $y $(echo cmd subst) $((3 + 2))
   | 1| |--- 2 ---|  3 |------ 4 ------| |-- 5 ---|
   ```

2. 检查第一个单词（echo）是否为关键字，例如 if 或 for。在这里不是，所以命令行不变继续处理。

3. 检查第一个单词（依然是 echo）是否为别名。在这里并不是，所以命令行不变，继续往下处理。

4. 扫描所有单词是否需要波浪号展开。在本例中，~+ 为 ksh93 与 bash 的扩展，等同于 $PWD，也就是当前的目录（这部分在 14.3.7 节介绍）。token 2 将被修改，处理继续如下：

   ```
   echo /tmp/x/${f}[12] $y $(echo cmd subst) $((3 + 2))
   | 1| |----- 2 -----|  3 |------ 4 ------| |-- 5 ---|
   ```

5. 下一步是变量展开：token 2 与 3 都被修改。这会产生：

   ```
   echo /tmp/x/f[12] a b $(echo cmd subst) $((3 + 2))
   | 1| |---- 2 ---| |3| |------ 4 ------| |-- 5 ---|
   ```

6. 再来要处理的是命令替换。注意，这里可递归引用列表里的所有步骤！在此例中，因为我们试图让所有的东西容易理解，因此命令替换修改了 token 4，结果如下：

   ```
   echo /tmp/x/f[12] a b cmd subst $((3 + 2))
   | 1| |---- 2 ---| |3| |-- 4 --| |-- 5 ---|
   ```

7. 现在执行算术替换。修改的是 token 5，结果是：

   ```
   echo /tmp/x/f[12] a b cmd subst 5
   | 1| |---- 2 ---| |3| |-- 4 --| 5
   ```

8. 前面所有的展开产生的结果，都将再一次被扫描，看看是否有 $IFS 字符。如果有，则它们是作为分隔符（separator），产生额外的单词。例如，两个字符 $y 原来是组

成一个单词，但展开式“a-空格-b”，在此阶段被切分为两个单词：a与b。相同方式也应用于命令替换$(echo cmd subst)的结果上。先前的token 3变成了token 3与4。先前的token 4则成了token 5与6。结果则为：

```
echo /tmp/x/f[12] a b cmd subst 5
| 1| |---- 2 ---| 3 4 |5| | 6 | 7
```

9. 最后的替换阶段是通配符展开。token 2变成了token 2与3，结果如下：

```
echo /tmp/x/f1 /tmp/x/f2 a b cmd subst 5
| 1| |-- 2 --| |-- 3 --| 4 5  6  | 7 | 8
```

10. 这时，Shell已经准备好要执行最后的命令了。它会去寻找echo。正好ksh93与bash里的echo都已内建到Shell中。

11. Shell实际执行命令。首先执行>out的I/O重定向，再调用内部的echo版本，显示最后的参数。

这是最后的结果：

```
$ cat out
/tmp/x/f1 /tmp/x/f2 a b cmd subst 5
```

7.8.1 eval语句

eval语句是在告知Shell取出eval的参数，并再执行它们一次，使它们经过整个命令行的处理步骤。这里有个例子可以让你了解eval究竟是什么。

eval ls会传递字符串ls给Shell执行，所以Shell显示当前目录下的文件列表。这个例子太简单了：字符串ls上没有东西需要被送出以让命令行处理两个步骤。所以我们来看这个：

```
listpage="ls | more"
$listpage
```

为什么Shell会把|与more看作ls的参数，而不是直接产生一页页的文件列表呢?这是由于在Shell执行变量时，管道字符出现在步骤5，也就是在它确实寻找管道字符之后(在步骤1)。变量的展开一直要到步骤8才进行解析。结果，Shell把|与more看作ls的参数，使得ls会试图在当前目录下寻找名为|与more的文件!

现在，想想eval $listpage吧。在Shell到达最后一步时，会执行带有ls、|与more参数的eval。这会让Shell回到步骤1，具有一行包括了这些参数的命令。在步骤1发现|后，将该行分割为两个命令：ls与more。每个要被处理的命令都以一般方式执行，最后的结果是在当前目录下已分页的文件列表。

7.8.2 subShell与代码块

还有两个其他的结构，有时也很有用：*subShell*与代码块（code block）。

subShell是一群被括在圆括号里的命令，这些命令会在另外的进程中执行（注4）。当你需要让一小组的命令在不同的目录下执行时，这种方法可以让你不必修改主脚本的目录，直接处理这种情况。例如，下面的管道是将某个目录树复制到另一个地方，在原始的V7 UNIX *tar*(1)的手册页里可以看到此例：

```
tar -cf - . | (cd /newdir; tar -xpf -)
```

左边的`tar`命令会产生当前目录的`tar`打包文件（archive），将它传送给标准输出。这份打包文件会通过管道传递给右边subShell里的命令。开头的`cd`命令会先切换到新目录，也就是让打包文件在此目录下解开。然后，右边的`tar`将从打包文件中解开文件。请注意，执行此管道的Shell（或脚本）并未更改它的目录。

代码块（code block）概念上与subShell雷同，只不过它不会建立新进程。代码块里的命令以花括号（{}）括起来，且对主脚本的状态会造成影响（例如它的当前目录）。一般来说，花括号被视为Shell关键字：意即它们只有出现在命令的第一个符号时会被识别。实际上：这表示你必须将结束花括号放置在换行字符（newline）或分号之后。例如：

```
cd /some/directory || {                                    代码块开始
    echo could not change to /some/directory! >&2          怎么了
    echo you lose! >&2                                     挖苦信息
    exit 1                                                 终止整个脚本
}                                                          代码块结束
```

I/O重定向也可套用到subShell（如先前的两个`tar`例子）与代码块里。在该情况下，所有的命令会从重定向来源读取它们的输入或传送它们的输出。表7-8简单说明subShell与代码块之间的差异。

表7-8：SubShell与代码块

结构	定界符	认可的位置	另外的进程
SubShell	( )	行上的任何位置	是
代码块	{ }	在换行字符、分号或关键字之后	否

要用subShell还是代码块需要根据个人的喜好而定。主要差异在于代码块会与主脚本共享状态。这么一来，`cd`命令会对主脚本造成影响，也会对变量赋值产生影响。特别是，

注4： POSIX的标准说法是"subShell环境"，即命令不需要真的在另外的进程中执行；然而，它们只是被禁止更换主脚本的环境（变量、目前目录等等）。ksh93会避免为subShell命令启动一个实际的进程（如果它可以的话）。大部分其他的Shell会产生一个单个进程。

代码块里的exit会终止整个脚本。因此，如果你希望将一组命令括起来执行，而不要影响主脚本，则应该使用subShell。否则，请使用代码块。

7.9 内建命令

Shell有为数不少的内建（built-in）命令。指的是由Shell本身执行命令，而并非在另外的进程中执行外部程序。POSIX更进一步将它们区分为“特殊（special）”内建命令以及“一般（regular）”内建命令。内建命令行参见表7-9，特殊内建命令以†标示。大部分这里列出的一般内建命令都必须内建于Shell中，以维持Shell正常地运行（例如read）。其他内建于Shell内的命令，则只是为了效率（例如true与false）。在此标准下，也为了更有效率，而允许内建其他的命令，不过所有的一般内建命令都必须能够以单个的程序被访问，也就是可以被其他二进制程序直接执行。像test内建命令就是为了让Shell的执行更有效率。

表7-9：POSIX的Shell内建命令

命令	摘要
:（冒号）注	不做任何事（只作参数的展开）
.（点号）注	读取文件并于当前Shell中执行它的内容
alias（别名）	设置命令或命令行的捷径（交互式使用）
bg	将工作置于后台（交互式使用）
break 注	从for、while或until循环中退出
cd	改变工作目录
command	找出内建与外部命令；寻找内建命令而非同名的函数
continue 注	跳到下一个for、while或until循环重复
eval 注	将参数当成命令行来处理
exec 注	以给定的程序取代Shell或为Shell改变I/O
exit 注	退出Shell
export 注	建立环境变量
false	什么事也不做，指不成功的状态
fc	与命令历史一起运行（交互式使用）
fg	将后台工作置于前台（交互式使用）
getopts	处理命令行的选项
jobs	列出后台工作（交互式使用）
kill	传送信号

表 7-9：POSIX 的 Shell 内建命令（续）

命令	摘要
newgrp	以新的 group ID 启动新 Shell（已过时）
pwd	打印工作目录
read	从标准输入中读取一行
readonly[注]	让变量为只读模式（无法指定的）
return[注]	从包围函数中返回
set[注]	设置选项或是位置参数
shift[注]	移位命令行参数
times[注]	针对 Shell 与其子代 Shell，显示用户与系统 CPU 时间的累计
trap[注]	设置信号捕捉程序
ture	什么事也不做，表示成功状态
umask	设置 / 显示文件权限掩码（mask）
unalias	删除别名定义（交互式使用）
unset[注]	删除变量或函数的定义
wait	等待后台工作完成

注： bash 的 source 命令（来自于 BSD C Shell）是等同于点号命令。

特殊内建命令与一般内建命令的差别在于 Shell 查找要执行的命令时。命令查找次序是先找特殊内建命令，再找 Shell 函数，接下来才是一般内建命令，最后则为 $PATH 内所列目录下找到的外部命令。这种查找顺序让定义 Shell 函数以扩展或覆盖一般 sehll 内建命令成为可能。

这一功能最常用于交互式 Shell 中。举例来说，当你希望 Shell 的提示号能包含当前目录路径的最后一个组成部分。最简单的实现方式，就是在每次你改变目录时，都让 Shell 改变 PS1。你可以写一个自己专用的函数，如下所示：

```
# chdir --- 在改变目录时也更新 PS1 的个人函数

chdir () {
    cd "$@"                     实际更改目录
    x=$(pwd)                    取得当前目录的名称，传到变量 x
    PS1="${x##*/}\$ "           截断前面的组成部分后，指定给 PS1
}
```

这么做会有个问题：你必须在 Shell 下输入 chdir 而不是 cd，如果你突然忘了而输入 cd，你会在新的目录上，但提示号就不会改变了。基于这个原因，你可以写一个名为 cd 的函数，然后 Shell 就会先找到你的函数，因为 cd 是一般内建命令：

```
# cd --- 改变目录时更新 PS1 的个人版本
#         (无法如实运行，见内文)

cd () {
    cd "$@"                    真的改变目录了吗！？
    x=$(pwd)                   取得当前目录的名称，并传给变量 x
    PS1="${x##*/}\$ "          截断前面的组成部分后，指定给 PS1
}
```

这里有一点美中不足：Shell函数如何访问真正的cd命令？这里所显示的cd "$@"只会再次调用此函数，导致无限递归。我们需要的是转义策略，告诉Shell要避开函数的查找并访问真正的命令。这正是内建命令command的工作，见例7-4。

例7-4：变更目录时更新PS1

```
# cd --- 改变目录时更新 PS1 的私人版本

cd () {
    command cd "$@"            实际改变目录
    x=$(pwd)                   取得当前目录的名称，传递给变量 x
    PS1="${x##*/}\$ "          截断前面的组成部分，指定给 PS1
}
```

command

语法

command [-p] **program** [**arguments** ...]

用途

在查找要执行的命令时，为了要避开Shell的包含函数。这允许从函数中访问与内建命令同名的命令的内建版本。

主要选项

-p

当查找命令时，使用$PATH的默认值，保证找到系统的工具。

行为

command会通过查阅特殊的与一般的内建命令，以找出指定的*program*，并沿着$PATH查找。使用-p选项，则会使用$PATH的默认值，而非当前的设置。

如果*program*为特殊内建命令，则任何的语法错误都不会退出Shell，且任何前置的变量指定在命令完成后，即不再有效。

警告

command非特殊内建命令。将函数命名为command的Shell程序设计人员可能会觉得很失望吧！

POSIX标准为特殊内建命令提供两个附加特性：

- 特殊内建工具语法上的错误，会导致Shell执行该工具时退出，然而当语法错误出现在一般内建命令时，并不会导致Shell执行该工具时退出。如果特殊内建工具遇到语法错误时不退出Shell，则它的退出值应该非零。
- 以特殊内建命令所标明的变量指定，在内建命令完成之后仍会有影响；这种情况不会发生在一般内建命令或其他工具的情况下。

第二项需要解释一下，我们先前在6.1.1节里曾提及，你可以在命令前面指定一个变量赋值，且变量值在被执行命令的环境中不影响当前Shell内的变量或是接下来的命令：

```
PATH=/bin:/usr/bin:/usr/ucb awk '...'
```

然而，当这样的指定用于特殊内建命令时，即使用在特殊内建命令完成之后，仍然会有影响。

表7-9列出本章到目前为止尚未介绍的命令。这些命令中的大部分对于Shell脚本而言为特殊情况或不相关情况，不过我们在这里还是作一些简介，让你了解它们在做什么以及使用它们的时机：

`alias`、`unalias`

分别用于别名的定义与删除。当命令被读取时，Shell会展开别名定义。别名主要用于交互式Shell；例如`alias 'rm=rm -i'`，指的是强制`rm`在执行时要进行确认。Shell不会作递归的别名展开，因此此定义是合法的。

`bg`、`fg`、`jobs`、`kill`

这些命令用于工作控制，它是一个操作系统工具，可将工作移到后台执行，或由后台执行中移出。

`fc`

是“fix command”的缩写，该命令设计用来在交互模式下使用。它管理Shell之前已存储的执行过的命令历史，允许交互式用户再次调用先前用过的命令，编辑它以及再重新执行它。

这条命令原先是在`ksh`下开发的，提供像BSD C Shell —— `csh`里的“`!-history`”这样的机制。不过`fc`现已被`ksh`、`bash`以及`zsh`所提供的交互式命令行编辑功能所取代。

`times`

该命令会打印由Shell及所有子进程所累积执行迄今的CPU时间。它对于日常的脚本编写不是那么有用。

umask

用来设置文件建立时的权限掩码，详见附录 B 里的说明。

剩下的两个命令在脚本中是用得到的。第一个是用来等待后台程序完成的 wait 命令。如未加任何参数，wait 会等待所有的后台工作完成；否则，每个参数可以是后台工作的进程编号（process ID），见 13.2 节，或是工作控制的工作规格。

最后，.（点号）也是非常重要的命令。它是用来读取与执行包含在各别文件中的命令。例如，当你有很多个 Shell 函数想要在多个脚本中使用时，正确方式是将它们放在各自的库文件里，再以点号命令来读取它们：

```
. my_funcs              # 在函数中读取
```

如指定的文件未含斜杠，则 Shell 会查找 $PATH 下的目录，以找到该文件。该文件无须是可执行的，只要是可读取的即可。

注意：任何读进来的（read-in）文件都是在当前 Shell 下执行。因此，变量赋值、函数定义，以及 cd 的目录变更都会有效。这和简单地执行各自的 Shell 脚本是很不同的，后者在个别的进程中执行，且完全不对当前的 Shell 有任何影响。

7.9.1 set 命令

set 命令可以做的事相当广泛（注 5）。就连使用的选项语法也与众不同，这是 POSIX 为保留历史的兼容性而存在的命令。也因为这样，这个命令有点难懂。

set 命令最简单的工作就是以排序的方式显示所有 Shell 变量的名称与值。这是调用它时不加任何选项与参数时的行为。其输出是采用 Shell 稍后可以重读的形式 —— 包含适当的引号。这个想法是出自 Shell 脚本有可能需要存储它的状态，在之后会通过 .（点号）命令恢复它。

set 的另一项任务是改变位置参数（$1、$2 等）。使用 -- 的第一个参数来结束设置它自己的选项，则所有接下来的参数都会取代位置参数，即使它们是以正号或负号开头。

注 5： 它因此违反了做好一件事的软件设计原则。这是因为，Steven Bourne 希望能避免有太多的保留命令内置在 Shell 里。

set

语法

```
set
set -- [ arguments ... ]
set [ -short-options ] [ -o long-option ] [ arguments ... ]
set [ +short-options ] [ +o long-option ] [ arguments ... ]
set -o
set +o
```

用途

为了打印当前Shell的所有变量名称及其值、为了设置或解除设置Shell选项的值（可改变Shell行为的方式），以及为了改变位置参数的值。

主要选项

见内文。

行为

- 无选项或参数，则以Shell稍后可读取的形式来打印所有Shell变量的名称与值。
- 选项为 `--` 及参数，则以提供的参数取代位置参数。
- 开头为 `-` 的短选项，或以 `-o` 开头的长选项，则可打开特定的Shell选项。额外的非选项（nonoption）参数可设置位置参数。详见内文。
- 以 `+` 开头的短选项，或以 `+o` 开头的长选项，则可关闭特定的Shell选项。详见内文。
- 单一的 `-o` 可以一种不特别指定的格式打印Shell选项的当前设置。ksh93与bash都会打印排序后的列表，其中每一行是一个选项名称与单词on或off：

```
$ set -o                     来自 bash
allexport       off
...
```

- 单一的 `+o` 则是显示Shell选项的当前设置，其采用Shell之后可以重读的方式，以获得选项设置的相同设置。

警告

除了表7-10所列的之外，实际的Shell还拥有其他额外的短与长选项，第14章有详细介绍。如果可移植性对你来说很重要，请不要用它们。

有些 `/bin/sh` 版本完全不认得 `set -o` 的用法。

最后，set 被用来打开或停用 Shell 选项（Shell option），指的是改变 Shell 行为模式的内部设置。这里就是复杂的地方了，从历史来看，Shell 选项是以单个字母来描述，以负号打开并且以正号关闭。POSIX 另加入了长选项，打开或关闭分别是用 -o 或 +o。每个单个字母选项都有相对应的长名称选项。表 7-10 列出部分选项，并简短说明了它们的功能。

表 7-10：POSIX Shell 选项

短选项	o 形式	说明
-a	allexport	输出所有后续被定义的变量。
-b	notify	立即显示工作完成的信息，而不是等待下一个提示号。供交互式使用。
-C	noclobber	不允许 > 重定向到已存在的文件。>| 运算符可使此选项的设置无效。供交互式使用。
-e	errexit	当命令以非零状态退出时，则退出 Shell。
-f	noglob	停用通配符展开。
-h		当函数被定义（而非当函数被执行）时，寻找并记住从函数体中被调用的命令位置（XSI）。
-m	monitor	打开工作控制（默认是打开的）。供交互式使用。
-n	noexec	读取命令且检查语法错误，但不要执行它们。交互式 Shell 被允许忽略此选项。
-u	nounset	视未定义的变量为错误，而非为 null。
-v	verbose	在执行前先打印命令（逐字打印）。
-x	xtrace	在执行前先显示命令（在展开之后）。
	ignoreeof	不允许以 Ctrl-D 退出 Shell。
	nolog	关闭函数定义的命令历史记录功能。
	vi	使用 vi 风格的命令行编辑。供交互式使用。

你可能感到意外的地方是：set 并非用来设置 Shell 变量（不像 BSD C Shell 里的相同命令那样）。该工作是通过简单的 *variable=value* 指定实现的。

注意： 尽管不是 POSIX 的一部分，set -o emacs 命令还是在很多 Shell 中实现出来（ksh88、ksh93、bash、zsh）。如果你已习惯使用 emacs，则使用此命令可让你的单行小窗口编辑器可以接受 emacs 命令，以与你的 Shell 历史一起运行。

特殊变量$-是一个字符串，表示当前已打开的Shell选项。每个选项的短选项字母会出现在字符串中（假如该选项是打开的话）。这可被用来测试选项设置，像这样：

```
case $- in
*C*)    ...            启用noclobber选项
        ;;
esac
```

警告： 值得玩味的是，当POSIX标准可让用户存储与恢复Shell变量与捕捉（trap）的状态时，它却没有统一的方式来存储函数定义的列表供而后再利用。这似乎是该标准的小疏失，我们将于14.1节中作说明。

7.10 小结

read命令会读取行并将数据分割为各个字段，供赋值给指明的Shell变量。搭配-r选项，可控制数据要如何被读取。

I/O重定向允许你改变程序的来源与目的地，或者将多个程序一起执行于subShell或代码块里。除了重定向到文件和从文件重定向之外，管道还可以用于将多个程序连接在一起。嵌入文件则提供了行内输入。

文件描述符的处理是基本操作，特别是文件描述符1与2，会重复地用在日常的脚本编写中。

printf是一个深具灵活性，但有点复杂的命令，用途是产生输出。大部分时候，它可以简单地方式使用，但其实它的力量有时候是很有必要性且深具价值的。

Shell会执行许多的展开（或替换）在每个命令行的文字上：波浪号展开式（如果有支持）与通配符、变量展开、算术展开及命令替换。通配符现已包含POSIX字符集，用来针对文件名内的字符进行独立于locale的匹配。为了使用上方便，点号文件并未包含在通配符展开中。变量与算术展开于第6章已做过说明。命令替换有两种形式：`...`为原始形式，而$(...)为较新、较好写的形式。

引用会保护不同的源代码元件，免于被Shell作特殊处理。单个的字符可能会以前置反斜杠的方式引用使用。单引号会保护所有括起来的字符；引号括起的所有文字都不作处理，且你不可以将单引号内嵌到以单引号引用的文字内。双引号则是组合括起来的项目，从而视为单一的单词或参数，但是变量、算术与命令替换仍旧应用到内容中。

eval命令的存在是为了取代一般命令行替换与执行顺序，让Shell脚本可以动态地构建

命令。这个功能很好用，但得小心使用。因为 Shell 拥有这么多种的替换功能，花点时间了解 Shell 在执行输入行时的顺序绝对是很有帮助。

subShell与代码块是组化命令的两种选择。它们的用意各有不同，你可以根据需求选用。

内建命令的存在是因为它们要改变 Shell 内部状态且必须是内建的（例如 cd），有些则是为了效率（例如 test）。命令查找顺序允许函数在一般内建命令之前先被找到，结合 command命令，则可以编写一个能使内建命令生效的Shell函数。在所有内建命令里，set 命令是最复杂的。

第8章

产生脚本

在本章中，我们将进一步处理更复杂的工作。我们认为这里举出的例子都是一般用得到的工具，它们每一个都是全然不同的，且在大多数的UNIX工具集里也没有。

本章中的程序，包括命令行参数分析、在远程主机上运算、环境变量、工作记录、并行处理、使用eval的运行时语句（runtime statement）执行、草稿文件（scratch file）、Shell函数、用户定义初始文件，以及安全性议题考虑的范例。程序会运用Shell语言里的重要语句，并展现传统UNIX Shell脚本编写风格。虽然我们是为了这本书而开发的这些程序，但这些程序的基础稳固，经得起时间考验，你可以在日常工作上使用它们。

8.1 路径查找

有些程序支持在目录路径上查找输入文件，有点像UNIX Shell查找以冒号隔开的目录列表，列在PATH内，以找出可执行的程序。这对用户来说很方便，他只要提供较短的文件名，且不需要知道它们在文件系统里的位置。UNIX并未提供任何特殊命令或系统调用，可在查找路径下寻找文件，即使在很久以前有其他操作系统支持这种功能。幸好，要作路径查找不难，只要用对工具就可以。

与其实现路径查找以寻找特定程序，不如做一个新的工具，以环境变量名称为参数，而环境变量名称展开是预期的查找路径，后面接着零个或更多的文件模式，并报告匹配文件的位置。我们的程序将在其他需要路径查找支持的软件中，成为一般性工具（这也就是我们先前在第1章提到的"构建特定工具前，先想想"的原则）。

有时你得知道路径下的某个文件是不是不止一个，因为当路径下存有不同的版本时，你可能需要调整路径，控制要找到的版本。我们的程序会为用户提供一个命令行选项，以选择报告第一个找到的文件还是报告所有找到的文件。另外，根据用户要求提供一个可

识别的版本编号，变成软件的标准实现，而且必须提供简短的在线帮助，让用户不需要在每次使用程序时，都重读程序的使用手册才能想起选项名称。我们的程序当然也提供这样的功能。

完整程序将在例8-1给出，因为这个程序很长，我们在此将它展现为伪码程序（semiliterate program），是为了注释与描述Shell程序代码的各个片段的顺序，以便说明。

我们从一般的介绍性注释块开始。首行为识别程序的神奇行，/bin/sh用来执行脚本。注释块后，接的是程序行为的描述，并说明使用方法：

```
#! /bin/sh -
#
# 在查找路径下寻找一个或多个原始文件或文件模式，
# 查找路径由一个指定的环境变量所定义。
#
# 标准输出产生的结果，通常是在查找路径下找到的每个文件之第一个实体的完整路径，
# 或是“filename: not found”的标准错误输出。
#
# 如果所有文件都找到，则退出码为 0，
# 否则，即为找不到的文件个数 - 非零值
#  (Shell 的退出码限制为 125)。
#
#
# 语法 :
#       pathfind [--all] [--?] [--help] [--version] envvar pattern(s)
#
# 选项 --all 指的是寻找路径下的所有目录，
# 而不是找到第一个就停止。
```

在网络的环境下，安全性一直是必须慎重考虑的问题。其中有一种攻击Shell脚本的方式，是利用输入字段分隔字符：IFS，它会影响Shell接下来对输入数据解释的方式。为避免此类型的攻击，部分Shell仅在脚本执行前，将IFS重设为标准值；其他则导入该变量的一个外部设置。我们则是将自己做的预防操作放在脚本的第一步：

```
IFS='
 '
```

很难在屏幕上或是显示的页面上看出位于引号里的内容：它是具有三个字符的字符串，包括换行字符（newline）、空格，以及定位字符（tab）。IFS的默认值为：空格、定位字符、换行字符，不过如果我们以这种方式编写，那些会自动修剪空白的编辑器可能会将结尾的空白截去，让字符串的值减少成只有一个换行字符。比较好的方式应该是更严谨的使用转义字符，例如IFS="\040\t\n"，可是Bourne Shell并不支持这样的转义符。

在我们重新定义IFS时有一点请特别留意。当"$*"展开以恢复命令行时，IFS值的第一个字符，会被当成字段分隔符。我们在这个脚本里不使用$*，所以重新安排IFS内的字符不会有影响。

另一种常见的安全性攻击，则是欺骗软件，它执行非我们所预期的命令。为了阻断这种攻击，我们希望调用的程序是可信任的版本，而非潜伏在用户提供的查找路径下的欺骗程序，因此我们将 PATH 重设为一个最小值，以存储初始值供以后使用：

```
OLDPATH="$PATH"

PATH=/bin:/usr/bin
export PATH
```

export 语句是这里的关键：它可以确保所有的子进程继承我们的安全查找路径。

程序代码接下来是 5 个以字母顺序排列的简短函数。

第 1 个函数为 error() 函数，在标准错误输出上显示其参数，再调用一个函数（此部分稍后有说明），不返回：

```
error()
{
    echo "$@" 1>&2
    usage_and_exit 1
}
```

第 2 个函数 usage() 会写出简短信息，显示程序的使用方式，并返回给它的调用者。需留意的是：这个函数需要程序名称，但不是以直接编码模式（hardcode），它是从变量 PROGRAM 取得程序名称，这个变量设置为程序被调用的名称。这可以让安装程序在重新命名程序时，无须修改程序代码，这常发生在与已安装的程序名称产生冲突时，也即具同样名称但具有不同用途的时候。函数本身很简单：

```
usage()
{
    echo "Usage: $PROGRAM [--all] [--?] [--help] [--version] envvar pattern(s)"
}
```

第 3 个函数 usage_and_exit()，会产生语法信息，并以它的单一参数所提供的状态码退出：

```
usage_and_exit()
{
    usage
    exit $1
}
```

第 4 个函数 version() 是在标准输出上显示程序版本编号，并返回给它的调用者。如同 usage()，它是使用 PROGRAM 取得程序名称：

```
version()
{
```

```
        echo "$PROGRAM version $VERSION"
    }
```

最后一个函数warning()会在标准错误上显示它的参数，并对变量EXITCODE加1，可记录已发出的警告信息的数目，并返回给调用者：

```
    warning()
    {
        echo "$@" 1>&2
        EXITCODE=`expr $EXITCODE + 1`
    }
```

7.6.3节对expr有较深入的探讨。在这里，它的语法是Shell为变量增值的习惯方式。较新版的Shell则允许更简单的形式EXITCODE=$((EXITCODE + 1))，不过还是有相当多系统不认得该POSIX用法。

即使这个程序很短，我们其实完全不需要写函数，除了避免程序代码重复之外，其实它可以隐藏不相关的细节，这是良好的程序实现方式：告知我们正要做什么，而不是说明我们要如何作。

这时我们已经到达运行时的第一条被执行的语句了。先初始化5个变量，以记录选项的选择、用户提供的环境变量名称、退出码、程序名称以及程序版本编号：

```
    all=no
    envvar=
    EXITCODE=0
    PROGRAM=`basename $0`
    VERSION=1.0
```

在我们的程序里，将遵循小写（字母）变量为本地函数或主程序代码体所使用，而大写变量则被整个程序全局性地共享。这里给all变量一个字符串值，而非一个数字，是因为这样可以让程序更清楚，而对运行时资源消耗的影响也微乎其微。

注意： basename命令是从完整路径名称中取出文件名称部分的传统工具，它会截断第一个参数的开头所有字符，一直到最后一个斜杠（含），再报告剩余字符到标准输出上：

```
$ basename resolv.conf                     产生仅含文件名的结果
resolv.conf
$ basename /etc/resolv.conf                产生仅含文件名的结果
resolv.conf
```

Bourne Shell的后代版本都提供模式匹配运算符，如第6章表6-2所示，可用来完成此目的，但basename为原始命令，所以可在所有Shell下运行。

使用第二个参数来表示文件名结尾，则basename从它的结果中截断任何匹配的结尾：

```
$ basename /etc/resolv.conf .conf        报告无结尾的文件名
resolv
$ basename /etc/resolv.conf .pid         仅报告文件名
resolv.conf
```

虽然 basename 的第一个参数通常是路径名称，但 basename 只简单地认为它是一个文字字符串，不需要、也不会检查它是否为真正存在的文件。

如省略参数，或参数为空字符串，则 basename 的行为由运行时定义。

接下来的大型代码块，是在所有的 UNIX 程序里典型的命令行参数解析：当我们有一个参数时（由参数计数值 $# 决定，且必须大于零），会根据参数的字符串值来选择 case 语句的程序块，处理该参数：

```
while test $# -gt 0
do
    case $1 in
```

case 选择器在不同环境下可能有不同的解读方式。GNU 程序风格鼓励使用长的、描述性的选项名称，而不是长久以来用于 UNIX 里的那套旧的、隐秘式的单一字符选项。后者简洁式的做法，在选项数很少且程序使用频繁的时候还让人可以接受，如果不是这种情况，描述性的名称会比较好，用户只需要提供足够的信息 —— 不要和其他选项重复即可。然而，当有其他程序提供相同的选项时，则应避免这种简略用法，这么做才能让用户更容易了解程序，确保日后程序新版本加入新选项时，不会产生意外的结果。

在 Shell 语言里，没有一种简单的方式来通过给名称添加明确的前缀来指定该名称可以与长名称进行匹配，所以我们必须提供所有的替代用法。

有时长选项名称会通过前置两个连字号经编辑后纳入旧式程序中，这是为了与原始选项区分。以新式程序代码来说，我们允许一个或两个连字号，它可以通过重复 case 选择器里的缩写并加入额外连字号，完成程序改版。

我们也许会使用通配字符编写 case 选择器以匹配：--a* | -a*)。但我们认为这是无法接受的松散实现方式，因为它允许匹配和那些写出来的名称全然不同的名称。

针对 --all 选项，我们只要通过把变量 all 重新设置为 yes 来记录找到选项的事实即可：

```
--all | --al | --a | -all | -al | -a )
    all=yes
    ;;
```

在每个 case 块之后的双分号是必备的 —— 除了最后一个 case 以外。我们也可以将这段写得更紧凑些：

```
    --all | --al | --a | -all | -al | -a ) all=yes ;;
```

上面这种方式，当双分号出现在它们自己的行上时，比较容易验证所有的情况都被适当地终止，而且对于在块里附加额外的语句也较为容易。适度使用缩进编辑则有助于程序的解读，还能强调程序逻辑结构，这几乎在所有程序语言里都适用。

以GNU处理--help需求的惯例是：在标准输出上，显示如何使用程序的简短摘要，且立即以成功状态码退出（在POSIX与UNIX都为0）。对大型程序而言，摘要应包含每个选项的简短说明，不过我们的程序很小，不需要额外的说明。由于问号?是Shell的通配字符，所以我们必须在case选择器里，以引号框起来：

```
    --help | --hel | --he | --h | '--?' | -help | -hel | -he | -h | '-?' )
        usage_and_exit 0
        ;;
```

同样地，GNU惯例也会针对--version选项，在标准输出上产生一行（通常是这样）报告结果，并立即以成功状态退出。同样的情形也应用到其他种类的状态要求选项（可能出现在大型程序里）上，例如--author、--bug-reports、--copyright、--license、--where-from等：

```
    --version | --versio | --versi | --vers | --ver | --ve | --v | \
    -version | -versio | -versi | -vers | -ver | -ve | -v )
        version
        exit 0
        ;;
```

case选择器-*)会匹配剩下的所有选项：我们会在标准错误输出上报告非法选项，并调用usage()函数，提醒用户用法为何，再立即以失败状态码（1）退出：

```
    -*)
        error "Unrecognized option: $1"
        ;;
```

标准错误输出与标准输出的差异在各软件间都不尽相同，且在交互模式下使用命令时，用户不会感觉到差异，因为两者数据流都流到相同的显示设备。如果程序为过滤器的角色，则错误信息与状态报告，像--help与-version选项所产生的输出，都应流至标准错误输出，这么一来才不致让管道有混乱的数据出现；否则，状态报告会流到标准输出。由于状态报告是GNU世界近期的新产物，程序实现仍在进化中，标准尚未出现。不管是POSIX还是传统的UNIX文件似乎都未对此议题做出任何说明。

最后一个case选择器*)是匹配上述选项以外的所有状态。它有点类似C、C++以及Java语言里switch语句的default选择器，包括它的做法一直都是个不错的构想，即便它的内容是空的，也能向读者说明：所有可能出现的状态我们都考虑到了。这里，匹配指

出我们已经处理完所有的选项，所以可以退出循环。由于我们已处理完所有可能的情况，所以这里用终结关键字来结束case语句：

```
        *)
            break
            ;;
        esac
```

我们现已经到了选项循环的最后。正好在它的最后一条语句之前，我们使用shift以抛弃现在已被处理的第一个参数，且能移到参数列表的下一个项目。这么做可确保：当参数计数 $# 到达零时，最终的循环会结束：

```
        shift
    done
```

从循环中退出时，所有选项都已处理，而且参数列表里剩下的就是环境变量名称以及要寻找的文件了。我们将变量名称存储在envvar中，且如果至少还有一个参数留下来时，我们就丢弃第一个参数：

```
    envvar="$1"
    test $# -gt 0 && shift
```

剩下来的参数可以 "$@" 取得，我们避免将它们存储在变量内，例如 files="$@"，因为文件名中如果有空格将无法正确地被处理：内嵌的空格将成为参数的分隔符。

因为有可能用户提供的环境变量是 PATH，我们为了安全性因素会重设，这时我们会检查该变量，并适当地更新 envvar：

```
    test "x$envvar" = "xPATH" && envvar=OLDPATH
```

开头的 x 是常见的：此处它是为了避免变量的展开（如果此展开是以一个连字号开头）与 test 的选项相混淆。

至此，所有参数都已处理，我们要进入最棘手的部分：使用Shell的 eval 语句。我们在 envvar 里已经拥有了环境变量的名称，可以 "$envvar" 取得，但我们现在要的是它的展开。我们也想要将冒号分隔字符转换成一般的空白分隔字符。如果MYPATH为用户所提供的名称，我们便会构建参数字符串 '${'"$envvar"'}'，也就是 Shell 展开为 '${MYPATH}' 的等同物。两边的单引号是为了避免它更进一步地展开。该字符串之后会传给 eval，eval 会将其视为两个参数：echo 与 ${MYPATH}。eval 在环境下寻找 MYPATH，假设找到 /bin:/usr/bin:/home/jones/bin，就执行展开的命令：echo /bin:/usr/bin:/home/jones/bin，接着将 /bin:/usr/bin:/home/jones/bin 传给管道直到 tr 命令将冒号转换空格字符，会产生 /bin /usr/bin /home/jones/bin。

两边的反引号（或是现代Shell使用的$(...)）将其转换为指定给dirpath的值。在此不输出eval的任何错误信息，我们将其传送到/dev/null：

```
dirpath=`eval echo '${'"$envvar"'}' 2>/dev/null | tr : ' ' `
```

我们花了这么长的段落，解释一个短短的dirpath语句的设置，你就可以知道这有多麻烦。简单来说，eval对程序语言来说是如虎添翼。

讨论完eval，接下来的程序就比较好了解了。首先是一些健康检查（sanity check），处理所有可能会引发问题的异常情况：每一个好程序都应该有这类的检测，以避免声名狼藉的垃圾信息输入、垃圾信息输出（garbage-in, garbage-out）症状。要留意的是最后一个对空文件列表进行的健康检查：并不输出任何错误报告。这是因为任何程序在处理列表时都可能会遇到空列表：如果没有事情要程序去做，则除了成功信息之外也没有必要输出任何报告：

```
# 针对错误状态进行健康检查
if test -z "$envvar"
then
    error Environment variable missing or empty
elif test "x$dirpath" = "x$envvar"
then
    error "Broken sh on this platform: cannot expand $envvar"
elif test -z "$dirpath"
then
    error Empty directory search path
elif test $# -eq 0
then
    exit 0
fi
```

接下来还有三个嵌套循环：最外面的是处理参数文件或模式，居中循环则处理查找路径下的目录，最里面循环匹配单一目录下的文件。按照这样的次序安排循环的目的，是为了在移到下一个文件之前，让每个文件都能完整地处理。相反的循环顺序，只会让用户更混淆，因为所有的文件报告都会混杂在一起。在开始中间循环前，我们将result设为空字符串，因为稍后我们将用它来决定是否找到任何东西：

```
for pattern in "$@"
do
    result=
    for dir in $dirpath
    do
        for file in $dir/$pattern
        do
```

在最里面的循环，test -f告诉我们$file是否存在以及是否是一个常规文件（如果是符号性连接也为真，因为它终究会指向一个常规文件）。如果是，则将它记录到

result中，并以echo命令报告到标准输出，而如果默认的报告仅应用第一个，则我们会跳出最内部与居中的循环。否则，循环会继续通过剩下的匹配文件，可能产生更多的输出结果：

```
            if test -f "$file"
            then
                result="$file"
                echo $result
                test "$all" = "no" && break 2
            fi
        done
    done
```

这个程序里，不需要在中间循环中测试$dir本身是否存在作为合法的目录，因为这部分已纳入最内部循环里的$file存在检测中。不过，具有一个较复杂的循环体，这样的测试会比较理想，也很容易做到，仅需一条单独语句：test -d $dir || continue。

当居中循环完成时，我们已经以$pattern查找过所有查找路径下的目录了，而result会握有最后匹配的名称，如果找不到匹配则为空。

我们测试展开式$result是否为空，如果是，则在标准错误输出上报告找不到的文件，并将EXITCODE里的错误计数值加1（在warning函数里），然后继续外层的循环以处理下一个文件：

```
        test -z "$result" && warning "$pattern: not found"
    done
```

在处理完外层的循环后，我们会在查找路径下的每个目录里进行每一个被要求的匹配，并准备好返回给调用程序。现在，只剩下一个小问题待解决：用户退出码的值被限制为范围0至125，如第6章的表6-5所示，这里我们将EXITCODE的值，设置为125：

```
    test $EXITCODE -gt 125 && EXITCODE=125
```

我们的程序可说是很完整：程序的最后一个语句，会返回退出状态给父进程，这是所有模范UNIX程序应该做的。在这种方式下，父进程可测试退出状态，以知道子进程是成功或者失败：

```
    exit $EXITCODE
```

例8-1里，我们给出了pathfind的完整内容，其中去掉了注释，你看到的就是Shell看到的程序。去掉注释与空行，整个程序约90行。

例8-1：查找输入文件的路径

```
#! /bin/sh -
#
```

```
# 在查找路径下寻找一个或多个原始文件或文件模式，
# 查找路径乃由特定的环境变量所定义。
#
# 标准输出所产生的结果，通常是查找路径下找到的每个文件之第一个实体的完整路径，
# 或是“filename: not found”的标准错误输出。
#
# 如果所有文件都找到，则退出码为 0，
# 否则，即为找不到的文件个数（非 0）
# （Shell 的退出码限制为 125）
#
#
# 语法：
#       pathfind [--all] [--?] [--help] [--version] envvar pattern(s)
#
# 使用 --all 选项时，在路径下的每个目录都会被查找，
# 而非停在第一个找到者。

IFS='
'

OLDPATH="$PATH"

PATH=/bin:/usr/bin
export PATH

error()
{
    echo "$@" 1>&2
    usage_and_exit 1
}

usage()
{
    echo "Usage: $PROGRAM [--all] [--?] [--help] [--version] envvar pattern(s)"
}

usage_and_exit()
{
    usage
    exit $1
}

version()
{
    echo "$PROGRAM version $VERSION"
}

warning()
{
    echo "$@" 1>&2
    EXITCODE=`expr $EXITCODE + 1`
}

all=no
envvar=
```

```
EXITCODE=0
PROGRAM=`basename $0`
VERSION=1.0

while test $# -gt 0
do
    case $1 in
    --all | --al | --a | -all | -al | -a )
        all=yes
        ;;
    --help | --hel | --he | --h | '--?' | -help | -hel | -he | -h | '-?' )
        usage_and_exit 0
        ;;
    --version | --versio | --versi | --vers | --ver | --ve | --v | \
    -version | -versio | -versi | -vers | -ver | -ve | -v )
        version
        exit 0
        ;;
    -*)
        error "Unrecognized option: $1"
        ;;
    *)
        break
        ;;
    esac
    shift
done

envvar="$1"
test $# -gt 0 && shift

test "x$envvar" = "xPATH" && envvar=OLDPATH

dirpath=`eval echo '${'"$envvar"'}' 2>/dev/null | tr : ' ' `

# 为错误情况进行健全检测
if test -z "$envvar"
then
    error Environment variable missing or empty
elif test "x$dirpath" = "x$envvar"
then
    error "Broken sh on this platform: cannot expand $envvar"
elif test -z "$dirpath"
then
    error Empty directory search path
elif test $# -eq 0
then
    exit 0
fi

for pattern in "$@"
do
    result=
    for dir in $dirpath
    do
```

```
        for file in $dir/$pattern
        do
            if test -f "$file"
            then
                result="$file"
                echo $result
                test "$all" = "no" && break 2
            fi
        done
    done
    test -z "$result" && warning "$pattern: not found"
done

# 限制退出状态是一般UNIX实现上的限制
test $EXITCODE -gt 125 && EXITCODE=125

exit $EXITCODE
```

本节最后展示程序的几个简单测试，使用UNIX系统都会有的一个查找路径——PATH。每个测试会包括退出码的显示：$?，所以我们可以验证错误的处理。首先，我们检测在线帮助（help）与版本（version）选项：

```
$ pathfind -h
Usage: pathfind [--all] [--?] [--help] [--version] envvar pattern(s)
$ echo $?
0

$ pathfind --version
pathfind version 1.0
$ echo $?
```

下一步，我们使用错误的选项并且遗漏参数，看看会出现什么样的错误报告：

```
$ pathfind --help-me-out
Unrecognized option: --help-me-out
Usage: pathfind [--all] [--?] [--help] [--version] envvar pattern(s)
$ echo $?
1

$ pathfind
Environment variable missing or empty
Usage: pathfind [--all] [--?] [--help] [--version] envvar pattern(s)
$ echo $?
1

$ pathfind NOSUCHPATH ls
Empty directory search path
Usage: pathfind [--all] [--?] [--help] [--version] envvar pattern(s)
$ echo $?
1
```

接下来我们提供一些无意义的文件名：

```
$ pathfind -a PATH foobar
foobar: not found
$ echo $?
1

$ pathfind -a PATH "name with spaces"
name with spaces: not found
$ echo $?
1
```

测试空的文件名列表：

```
$ pathfind PATH
$ echo $?
0
```

如果我们突然按Ctrl-C中断执行中的程序，看看会发生什么事：

```
$ pathfind PATH foo
^C
$ echo $?
130
```

退出码为128 + 2，是指标志数字2被捕捉，并中止程序。在此特定的系统上，它是INT标志，对应于键盘中断字符的交互式输入。

迄今为止，错误报告都像预期的那样出现。现在让我们寻找一个已知存在的文件，然后再试试 -a 选项：

```
$ pathfind PATH ls
/usr/local/bin/ls
$ echo $?
0

$ pathfind -a PATH ls
/usr/local/bin/ls
/bin/ls
$ echo $?
```

接下来，检查引号内通配字符模式的处理，其必须匹配我们已知存在的文件：

```
$ pathfind -a PATH '?sh'
/usr/local/bin/ksh
/usr/local/bin/zsh
bin/csh
/usr/bin/rsh
/usr/bin/ssh
```

然后再以同样方式匹配不存在的模式：

```
$ pathfind -a PATH '*junk*'
*junk*: not found
```

接下来是大型的测试：寻找此系统里的一些C与C++编译器：

```
$ pathfind -a PATH c89 c99 cc c++ CC gcc g++ icc lcc pgcc pgCC
c89: not found
c99: not found
/usr/bin/cc
/usr/local/bin/c++
/usr/bin/c++
CC: not found
/usr/local/bin/gcc
/usr/bin/gcc
/usr/local/gnat/bin/gcc
/usr/local/bin/g++
/usr/bin/g++
/opt/intel_cc_80/bin/icc
/usr/local/sys/intel/compiler70/ia32/bin/icc
/usr/local/bin/lcc
/usr/local/sys/pgi/pgi/linux86/bin/pgcc
/usr/local/sys/pgi/pgi/linux86/bin/pgCC
$ echo $?
3
```

一个awk的单命令行，可用来验证退出码计数逻辑的运行就像我们所预期的那样。我们尝试了150个不存在的文件，但退出码正确地限制在125：

```
$ pathfind PATH $(awk 'BEGIN { while (n < 150) printf("x.%d ", ++n) }' )
x.1: not found
...
x.150: not found

$ echo $?
125
```

我们最后的测试会验证标准错误输出以及标准输出是否都如预期的那样，方式是将两个数据流捕捉到两个独立的文件中，再显示它们的内容：

```
$ pathfind -a PATH c89 gcc g++ >foo.out 2>foo.err
$ echo $?
1

$ cat foo.out
/usr/local/bin/gcc
/usr/bin/gcc
/usr/local/gnat/bin/gcc
/usr/local/bin/g++
/usr/bin/g++

$ cat foo.err
c89: not found
```

现在，我们可以说pathfind命令成功了。尽管有部分Shell向导仍可能发现里头有隐匿

的问题（注 1），而且无法替换大规模的测试，特别是具有非预期的输入时，例如附录B中“UNIX的文件里有什么？”部分注释所提到的模糊测试。理想上的测试，应结合有效参数与至少一个以上的无效参数。因为我们有三个主要选项的选择，每个都有几种缩写方式，所以就有(6 + 1) ×(10 + 1) × (14 + 1) = 1155种选项组合。每个组合都必须搭配0到3个（至少要三个）参数作测试。我们的程序在实现上对选项缩写的处理方式并无不同，所以只需要较少的必要测试。然而，当戴上测试帽子时，我们必须先将程序看作一个内容未知的黑盒子，但是仍有文件说明其特定的行为。之后，我们应再做不同思维的测试，潜入程序内部，了解它是怎么运行的，然后想出如何破坏它。而且测试的数据也要经过设计，要能够对程序的每一行进行彻底的测试，详尽的测试是相当冗长乏味的！

因为没有文件的软件可能是无法使用的软件，且因为很少有书籍会介绍如何编写使用手册，所以我们将 pathfind 的手册页放在附录 A。

pathfind确实是很有用的练习。除了它是个方便的新工具程序，而标准的GNU、POSIX以及UNIX工具集里都没有之外，它还拥有所有大多数UNIX程序的主要组成部分：参数解析、选项处理、错误报告以及数据处理。我们还说明了消除几个著名安全漏洞的三个步骤：加入 - 选项以终止起始的Shell命令行，立即设置 IFS及 PATH。该程序代码的好处是可以再利用，只需作一点点修改，例如：前置的注释标志、IFS与PATH的分配、5个辅助函数、处理参数的while与case语句，而且至少外部循环会遍历收集命令行上的文件。

作为一个练习，你可以开始考虑，是不是该为 pathfind 的这些扩展做一些改变：

- 将标准输出与标准错误输出的重定向存储到 /dev/null，并加上 --quiet 选项抑制所有输出，所以唯一会看到的便是指出是否找到匹配的退出码。这个好用的程序功能，在 cmp 的 -s 与 grep 的 -q 选项里已经有了。
- 加上 --trace 选项，将每个要测试的文件完整路径响应到标准错误输出。
- 加入 --test x选项，让test的 -f 选项可以置换为其他值，例如 -h（文件为符号性连接）、-r（文件是可读取的）、-x（文件是可具执行的）等。
- 让pathfind扮演过滤器功能：如果命令行上没有指定的文件名，则它应该自标准输入读取文件列表。这么做会对程序的架构与组织产生什么样的影响？
- 修补所有你找得到的安全漏洞，例如最新安全性公告所列的议题。

注 1： 有名的安全性漏洞包括了篡改输入字段分隔字符（IFS）；篡改查找路径，以恶意命令替换原可信的命令；暗中将反引号命令、Shell meta 字符以及控制字符（含NUL与换行符号）置入参数中；引发非预期的运行时中断；传送超过各种内部Shell资源限制长度的参数。

8.2 软件构建自动化

由于UNIX可运行在多种平台上，因此在构建软件包时，较常见的实现方式是从源代码开始安装，而非直接安装二进制包。大型的UNIX站点常由数个平台结合而成，对管理者而言，最冗长麻烦的工作就是将包安装在这些不同的系统上。而这正是自动化的好机会。很多软件开发人员，已直接采用GNU项目下所开发的软件包惯例，包括：

- 包以压缩存档文件`package-x.y.z.tar.gz`（或`package-x.y.z.tar.bz2`）的形式发布，文件解开后将出现在`package-x.y.z`目录下。
- 顶层`configure`脚本通常是由GNU的`autoconf`命令，通过`configure.in`或`configure.ac`文件里的规则列表自动产生。执行该脚本时，有时得加上一些命令行选项，便能产生定制的C/C++头文件，通常叫做`config.h`、衍生自`Makefile.in`（模板文件）的一个定制`Makefile`，并且有时候还会有其他文件。
- `Makefile`目标（target）的标准集已详述于《The GNU Coding Standards》中，有`all`（全部构建）、`check`（执行验证测试）、`clean`（删除不需要的中间文件）、`distclean`（恢复目录到它的原始发布），以及`install`（在本地系统上安装所有必需的文件）。
- 被安装的文件位于`Makefile`文件里`prefix`变量所定义的默认树状结构目录下，并且可在配置时使用`--prefix=dir`命令行选项进行设置，或是通过一个本地系统范围的定制文件提供。默认的prefix为`/usr/local`，但无权限的用户可能得使用`$HOME/local`，或是用`$HOME/`arch`/local`更好，其中`arch`是一条命令，它会显示定义平台的简短说明。GNU/Linux与Sun Solaris提供的是`/bin/arch`。在其他平台下，安装自己的实作程序时，通常只是使用简单的Shell脚本包装（wrapper），再搭配适当的`echo`命令。

接下来的工作就是建立脚本，它被给定一个包列表，在目前系统下的许多标准位置之一，找到它们的源分发，将它们复制到远程主机列表中的每一个，在那里解开它们，然后编译并使其成为合法可用状态。我们发现自动化安装步骤不是聪明的做法：构建日志必须先审慎地检查。

这个脚本必须让UNIX站点里的所有用户都能使用，所以我们不能在它里面内嵌有关特定主机的信息。我们假设用户已经提供两个定制文件：`directories`——列出要查找包分发文件的位置，以及`userhosts`——列出用户名称、远程主机名称、远程构建目录以及特殊环境变量。我们将这些及其他相关文件放在隐藏目录`$HOME/.build`下，以降低混乱的程度。然而，因为在同一个站点内所有用户下的来源目录列表可能都相类似，

所以我们包括一个合理的默认列表，就不再需要 directories 文件。

有时候构建要能够只在一般构建主机的子集主机上完成，或是使用不在一般位置里的存档（archive）文件，因此，脚本要能够在命令行上设置这些值。

我们在这里开发的脚本可以这样调用：

```
$ build-all coreutils-5.2.1 gawk-3.1.4              在所有主机上构建这两个包
$ build-all --on loaner.example.com gnupg-1.2.4    在指定主机上构建包
$ build-all --source $HOME/work butter-0.3.7       从非标准位置中构建包
```

这些命令其实做了很多事，下面我们大致列出，处理每个指定的软件包及在默认的或选定的构建主机上安装的步骤：

1. 在本地文件系统下寻找包分发文件。
2. 将分发文件复制到远程构建主机。
3. 初始化远程主机上的登录连接。
4. 切换到远程构建目录，并解开分发文件。
5. 切换到包构建目录并设置、构建与测试包。
6. 将初始化主机上的所有输出，分别为每个包与构建环境，记录在分开的日志文件中。

在远程主机上的构建以并行方式进行，所以安装执行所需要的总时间是以最慢的那台机器为基准，而不是把所有单个时间求和。对于动辄安装百种以上不同环境系统的我们来说，幸好有 build-all 程序，这也为包开发人员提供了一个不错的挑战。

build-all 脚本很长，所以我们分部分来展示，最后再显示完整程序代码于本章的例 8-2。开头使用一般的介绍性注释标头：

```
#! /bin/sh -
# 以并行处理的方式，在一台或多台构建主机上，建立一个或多个包。
#
# 语法 :
#       build-all [ --? ]
#                 [ --all "..." ]
#                 [ --cd "..." ]
#                 [ --check "..." ]
#                 [ --configure "..." ]
#                 [ --environment "..." ]
#                 [ --help ]
#                 [ --logdirectory dir ]
#                 [ --on "[user@]host[:dir][,envfile] ..." ]
```

```
#                    [ --source "dir ...." ]
#                    [ --userhosts "file(s)" ]
#                    [ --version ]
#                    package(s)
#
# 可选用的初始文件：
#       $HOME/.build/directories        list of source directories
#       $HOME/.build/userhosts          list of [user@]host[:dir][,envfile]
```

我们将字段分隔字符 IFS 初始化为“换行符号－空白－定位字符”：

```
IFS='
        '
```

接下来，将查找路径设置为一个有限制的列表，并用 export 使它成为全局性，这么一来在初始化主机上的所有子进程都可使用它：

```
PATH=/usr/local/bin:/bin:/usr/bin
export PATH
```

我们设置访问权限掩码（见附录 B 中关于默认权限的讨论），允许用户与组具有完整的访问权限，除此之外的其他人则仅具读取权限。组给定了完整访问权限，是因为在我们的部分系统中，有超过一个以上的系统管理者在处理软件安装，这些管理员都属于一个共同的可信任组。相同的掩码稍后在远程主机上也会需要，所以我们遵循程序惯例，以大写字母命名：

```
UMASK=002
umask $UMASK
```

它已经证明将工作的各部分委托给个别的函数处理会很方便，这么一来我们就能够将代码块限制在适当的大小了。程序里定义有 9 个这样的函数。但我们会在讨论到程序主要内容时再介绍。

我们需要一些变量，这些变量有很多一开始都是空值，以收集命令行设置：

```
ALLTARGETS=                     程序或make target 构建用
altlogdir=                      日志文件的另一个位置
altsrcdirs=                     来源文件的另一个位置
ALTUSERHOSTS=                   列出额外主机的文件
CHECKTARGETS=check              执行包测试的make target 名称
CONFIGUREDIR=.                  配置脚本的子目录
CONFIGUREFLAGS=                 配置程序的特殊标志
LOGDIR=                         本地目录，以保留日志文件
userhosts=                      在命令行上指定的额外构建主机
```

我们也需要参照多次 build-all 的初始化文件所在的目录，所以这里给它一个名称：

```
BUILDHOME=$HOME/.build
```

接下来是两个脚本，它们是在构建的开始与结尾在远程主机上的登录Shell内执行，以提供进一步的定制与日志文件的报告。这两者都克服了使用ksh或sh的登录Shell时在secure-Shell（ssh）上遭遇的问题：这两个Shell不会读取$HOME/.profile，除非它们是以登录Shell被启动，而且如果secure Shell是以命令行参数被调用，则它也不会安排任何处理，就像build-all一样：

```
BUILDBEGIN=./.build/begin
BUILDEND=./.build/end
```

正如例8-1的pathfind做法，设置最终退出码：

```
EXITCODE=0
```

没有默认的额外环境变量：

```
EXTRAENVIRONMENT=                    任何额外的环境变量都会传入
```

稍后会需要的程序名称，所以我们在这里存储其值与版本编号：

```
PROGRAM=`basename $0`                记住程序名称
VERSION=1.0                          记录程序版本编号
```

我们还在构建日志文件名中纳入时间戳，以DATEFLAGS的日期格式，取得以时间排序后的文件名。去除标点符号后，就是ISO 8601:2000所建议的格式（注2）。我们之后会在远程主机上以同样的方式调用date，所以我们希望这个复杂的日期格式，仅在一个地方定义：

```
DATEFLAGS="+%Y.%m.%d.%H.%M.%S"
```

在我们的站点，与远程主机通信使用的是secure Shell，而且scp与ssh两者我们都需要。仍沿用旧式不安全的远程Shell（remote Shell）的站点，则会更换为rcp与rsh。开发期间，我们设置这些变量为"echo scp"与"echo ssh"，这么做可以让日志记录我们做过的事，而不必真的去做它：

```
SCP=scp
SSH=ssh
```

对有些用户与系统配置文件设置而言，ssh会建立个别的加密通道（channel）供X Window System流量使用，而我们在软件构建过程中几乎完全用不到这样的功能，所以

注2： 见 *http://www.iso.ch.cate/d26780.html* 的《Date elements and interchange formats - Information interchange - Representation of dates and times》。该标准以YYYY-MM-DDThh:mm:ss或YYYYMMDDThhmmss的格式表示日期。为了可移植性，第一种格式的冒号不应出现在文件名里，而第二种格式，用户看起来比较吃力。

可以加入 -x 选项，关闭此功能，以降低启动时的负载，除非 SSHFLAGS 环境变量之设置提供了不同的选项集：

```
SSHFLAGS=${SSHFLAGS--x}
```

在初始文件中使用 Shell 风格的注释固然方便，不过我们还是可以使用 STRIPCOMMENTS 将它们删除，这是假设此注释字符并未出现在文件中：

```
STRIPCOMMENTS='sed -e s/#.*$//'
```

我们也需要一个过滤器，将数据流过滤为内缩状（比较好看的输出），并将换行字符置换为空格：

```
INDENT="awk '{ print \"\t\t\t\" \$0 }'"
JOINLINES="tr '\n' '\040'"
```

接下来是两个可选用的初始文件的定义：

```
defaultdirectories=$BUILDHOME/directories
defaultuserhosts=$BUILDHOME/userhosts
```

最后的初始化会设置来源目录的列表：

```
SRCDIRS="`$STRIPCOMMENTS $defaultdirectories 2> /dev/null`"
```

由于命令替换会将换行字符转换成空格且以空白字符凹入的方式排列，在初始化文件里的目录，可以写成一行一个或多个目录的方式。

如果用户定制文件不存在，STRIPCOMMENTS 会在 SRCDIRS 中产生一个空字符串，所以我们要测试这样的情况，并把 SRCDIRS 重新设置为合理的默认值列表，这个列表根据我们多年的使用经验得来：

```
test -z "$SRCDIRS" && \
    SRCDIRS="
            .
            /usr/local/src
            /usr/local/gnu/src
            $HOME/src
            $HOME/gnu/src
            /tmp
            /usr/tmp
            /var/tmp
        "
```

在行结尾处，|| 和 && 运算符后面接着一个反斜杠是 C-Shell 家族的要求，而这么作对 Bourne-Shell 家族也无碍。当前目录（.）为此列表的成员，因为我们可能就只是将要构建的包文件下载到一个任意位置。

现在初始化该做的事都照顾到了，所以我们已经准备好处理命令行选项。这个任务在所有 Shell 脚本下的处理方式都相同：while（当）参数仍在时，选择 case 语句的一个合适的分支来处理该参数，然后再 shift（移到）参数列表里的下一个参数，并继续循环。任何需要先消耗另一个参数的分支，会进行移位。正如我们之前做过的，我们允许单个与双个连字号的选项形式，而且我们也允许它们缩短为任何唯一性的字首：

```
while test $# -gt 0
do
    case $1 in
```

--all、--cd、--check，--configure 情况会存储接下来的参数，丢弃任何前一个存下来的值：

```
--all | --al | --a | -all | -al | -a )
    shift
    ALLTARGETS="$1"
    ;;

--cd | -cd )
    shift
    CONFIGUREDIR="$1"
    ;;

--check | --chec | --che | --ch | -check | -chec | -che | -ch )
    shift
    CHECKTARGETS="$1"
    ;;

--configure | --configur | --configu | --config | --confi | \
--conf | --con | --co | \
-configure | -configur | -configu | -config | -confi | \
-conf | -con | -co )
    shift
    CONFIGUREFLAGS="$1"
    ;;
```

--environment 选项可让用户在构建主机上，提供配置期环境变量的一次性设置，而无须要修改构建配置文件：

```
--environment | --environmen | --environme | --environm | --environ | \
--enviro | --envir | --envi | --env | --en | --e | \
-environment | -environmen | -environme | -environm | -environ | \
-enviro | -envir | -envi | -env | -en | -e )
    shift
    EXTRAENVIRONMENT="$1"
    ;;
```

--help 情况会调用我们尚未被显示的函数之一，并终止程序：

```
--help | --hel | --he | --h | '--?' | -help | -hel | -he | -h | '-?' )
    usage_and_exit 0
    ;;
```

--logdirectory情况也存储接下来的参数，丢弃任何已存储的值：

```
--logdirectory | --logdirector | --logdirecto | --logdirect | \
--logdirec | --logdire | --logdir | --logdi | --logd | --log | \
--lo | --l | \
-logdirectory | -logdirector | -logdirecto | -logdirect | \
-logdirec | -logdire | -logdir | -logdi | -logd | -log | -lo | -l )
    shift
    altlogdir="$1"
    ;;
```

altlogdir变量乃是当要被写入所有构建日志文件的默认目录不是我们所想要的时，用来作为指定目录的名称。

--on与--source情况只是累积参数，所以用户可以写成-s "/this/dir /that/dir"或-s /this/dir -s /that/dir：

```
--on | --o | -on | -o )
    shift
    userhosts="$userhosts $1"
    ;;

--source | --sourc | --sour | --sou | --so | --s | \
-source | -sourc | -sour | -sou | -so | -s )
    shift
    altsrcdirs="$altsrcdirs $1"
    ;;
```

由于altsrcdirs会以空格分隔列表的组成部分，所以含空格的目录名称可能无法正确地处理；请避免使用这样的名称。

--userhosts情况也为累积参数，但是具有检查另一个可选择的目录位置的额外功能，所以我们直接把工作移交给函数：

```
--userhosts | --userhost | --userhos | --userho | --userh | \
--user | --use | --us | --u | \
-userhosts | -userhost | -userhos | -userho | -userh | \
-user | -use | -us | -u )
    shift
    set_userhosts $1
    ;;
```

--version情况显示版本编号，并以成功状态码退出：

```
--version | --versio | --versi | --vers | --ver | --ve | --v | \
-version | -versio | -versi | -vers | -ver | -ve | -v )
    version
    exit 0
    ;;
```

下一个到最后一个情况会捕捉任何未被认可的选项，再以错误状态终止：

```
-*)
    error "Unrecognized option: $1"
    ;;
```

最后一个情况是匹配选项名称以外的任何东西，所以它必须是一个包名称，然后退出选项循环：

```
*)
    break
    ;;
esac
```

shift 会舍弃刚处理过的参数，然后继续下一个循环重复：

```
    shift
done
```

我们还需要一个邮件客户端程序报告日志文件的位置。遗憾的是：有些系统提供的是初级的mail命令，它不接受主题行，但另有mailx命令可以做这事。其他缺乏mailx的系统，在mail中却有支持主题行。也有其他系统两者功能都有，其中一个会连接到另一个。由于build-all必须在不做任何改变之下才能运行于所有UNIX版本上，所以我们不能把想用的邮件客户端程序名称直接编码(hardcode)。事实上，我们必须动态地查找它，使用我们在所有 UNIX 中所找到的列表：

```
for MAIL in /bin/mailx /usr/bin/mailx /usr/sbin/mailx /usr/ucb/mailx \
            /bin/mail /usr/bin/mail
do
    test -x $MAIL && break
done
test -x $MAIL || error "Cannot find mail client"
```

如果用户提供额外的来源目录，则我们会将它们置于默认列表的前端。取代默认列表的可能性不一定要有任何的值，所以我们不提供这样做的方式：

```
SRCDIRS="$altsrcdirs $SRCDIRS"
```

最终userhosts列表的正确设置是很复杂的，且需要解释。针对列表，这里有三个潜在的数据来源：

- 命令行--on选项，将它们的参数增加到userhosts变量。
- 命令行--userhosts选项会将文件(每一个包含零或多个构建主机描述)增加到ALTUSERHOSTS变量。
- defaultuserhosts变量包含了提供默认构建主机规格的文件之名称，只有当无命

令行选项提供它们时才会被使用。对大部分build-all的引用来说，该文件提供了完整的构建列表。

如果userhosts变量里有数据，则记录于ALTUSERHOSTS里的任何文件的内容，都会加进去，以获得最终的列表：

```
if test -n "$userhosts"
then
    test -n "$ALTUSERHOSTS" &&
        userhosts="$userhosts `$STRIPCOMMENTS $ALTUSERHOSTS 2> /dev/null`"
```

否则，如果userhosts变量为空，则仍有两种可能作法。如果ALTUSERHOSTS已设置，我们会保持原封不动，如果未设置，则将其设置为默认文件。接着，我们指定ALTUSERHOSTS里的文件的内容给userhosts变量，成为最终列表：

```
else
    test -z "$ALTUSERHOSTS" && ALTUSERHOSTS="$defaultuserhosts"
    userhosts="`$STRIPCOMMENTS $ALTUSERHOSTS 2> /dev/null`"
fi
```

在开始真正的操作之前，为确保我们至少有一个主机，有必要进行健康检查。虽然在此种情况下最里面的循环不会被执行，但我们还是要避免产生不必要的目录与日志文件。如果userhosts为空，则有可能是用户大意了，所以这时可以适时地提醒他这个程序的用法：

```
test -z "$userhosts" && usage_and_exit 1
```

最后是程序最外部的循环，用来处理包。如果参数列表为空，则Shell不会执行循环体，这正是我们要的。这个循环很大，所以我们一次只介绍几行即可：

```
for p in "$@"
do
```

在来源目录列表里，寻找包存档文件的工作委托find_package函数来做，它会将结果留在全局性变量：PARFILE（包存档文件；package archive file）里：

```
find_package "$p"
```

如PARFILE为空，则我们就把使用方式错误的信息发出到标准错误输出，再继续处理下一个包：

```
if test -z "$PARFILE"
then
    warning "Cannot find package file $p"
    continue
fi
```

另一方面，如果未提供日志目录，或有提供但不是目录或是不能写入，则我们会试图在包存档文件被找到的目录下，建立名为 logs 的子目录。如果找不到该目录，或无法写入，则我们会试着将日志文件放在用户的 $HOME/.build/logs 目录或临时目录内。我们倾向于尽量不要使用临时目录 /tmp，因为在开机后通常它的内容会消失，所以只将它作为没办法时的最后手段。

```
LOGDIR="$altlogdir"
if test -z "$LOGDIR" -o ! -d "$LOGDIR" -o ! -w "$LOGDIR"
then
    for LOGDIR in "`dirname $PARFILE`/logs/$p" $BUILDHOME/logs/$p \
                  /usr/tmp /var/tmp /tmp
    do
        test -d "$LOGDIR" || mkdir -p "$LOGDIR" 2> /dev/null
        test -d "$LOGDIR" -a -w "$LOGDIR" && break
    done
fi
```

注意： dirname 命令是与 8.1 节介绍过的 basename 命令一起的。dirname 会截断其参数里最后斜杠之后的所有字符，以从完整路径名称中恢复一个目录路径，并将结果显示到标准输出：

```
$ dirname /usr/local/bin/nawk                报告目录路径
/usr/local/bin
```

如果参数里没有斜杠，dirname 会产生一个点号（.）以表示目前目录：

```
$ dirname whimsical-name                     报告目录路径
.
```

dirname 就像 basename 一样，视其参数为单纯的文字字符串，而不会去检查目录是否真的存在于文件系统中。

如省略参数，则 dirname 的行为模式由运行时定义。

我们会告知用户日志文件建立于何处，并将该位置记录于电子邮件中，因为用户有可能在大型包构建完成之前就忘了日志文件的位置：

```
msg="Check build logs for $p in `hostname`:$LOGDIR"
echo "$msg"
echo "$msg" | $MAIL -s "$msg" $USER 2> /dev/null
```

主循环里的最后一步便是通过嵌套循环，以并行处理的方式，令每台远程主机开始构建现行包。我们再一次将大部分的工作交给函数来做。这也会结束最外部循环：

```
    for u in $userhosts
    do
        build_one $u
    done
done
```

build_one的调用为连续性的，所以我们可以更容易地识别出通信的问题。然而，它们在远程构建主机上起始的工作都以后台执行，所以 build_one 其实很快就完成。

现在，程序已完成它的工作。最后的语句是将累加状态码最高限定在125，并将状态码返回给调用者：

```
test $EXITCODE -gt 125 && EXITCODE=125
exit $EXITCODE
```

我们已将许多构建过程留在后台执行，将它们的输出信息不断累积在相关的日志文件里，并选择无论结果为何都会退出，所以 build-all 才得以执行得这么快。

有人可能会倾向于另一种设计方式，就是先不返回，一直等到所有后台进程都完成时再返回。要改成这种方式也很简单：只要在最后的 exit 语句之前插入以下这条语句：

```
wait
```

我们不觉得这种方式好，因为它会挂在终端窗口上直到所有构建完成为止，或是如果 build-all在后台执行，它的完成通知可能会混合许多其他的输出。如果它出现得太后面的话，可能还会找不到。

现在，我们对程序的运行已有大概的蓝图，该是审视隐藏在函数内的细节的时候了。我们将根据使用上的方便依次介绍。

usage 是一个简单的函数：在标准输出上打印短的帮助信息，使用嵌入文件（here document）形式，而非一连串 echo 语句：

```
usage()
{
    cat <<EOF
Usage:
        $PROGRAM [ --? ]
                        [ --all "..." ]
                        [ --cd "..." ]
                        [ --check "..." ]
                        [ --configure "..." ]
                        [ --environment "..." ]
                        [ --help ]
                        [ --logdirectory dir ]
                        [ --on "[user@]host[:dir][,envfile] ..." ]
                        [ --source "dir ..." ]
                        [ --userhosts "file(s)" ]
                        [ --version ]
                        package(s)
EOF
}
```

usage_and_exit 调用 usage，然后以提供的状态码作为它的参数而退出：

```
usage_and_exit()
{
    usage
    exit $1
}
```

version在标准输出上显示版本编号：

```
version()
{
    echo "$PROGRAM version $VERSION"
}
```

error在标准错误输出上显示它的参数以及用法信息，然后以错误状态码终止程序：

```
error()
{
    echo "$@" 1>&2
    usage_and_exit 1
}
```

warning在标准错误输出上显示其参数，在EXITCODE内加1（警告计数），并返回：

```
warning()
{
    echo "$@" 1>&2
    EXITCODE=`expr $EXITCODE + 1`
}
```

主程序代码最外部的循环是由调用find_package开始，该函数会循环遍历来源目录以寻找包，并处理我们还未提及的细节：

```
find_package()
{
    # Usage: find_package package-x.y.z
    base=`echo "$1" | sed -e 's/[-_][.]*[0-9].*$//'`
    PAR=
    PARFILE=
    for srcdir in $SRCDIRS
    do
        test "$srcdir" = "." && srcdir="`pwd`"

        for subdir in "$base" ""
        do
            # NB: update package setting in build_one() if this list changes
            find_file $srcdir/$subdir/$1.tar.gz  "tar xfz"    && return
            find_file $srcdir/$subdir/$1.tar.Z   "tar xfz"    && return
            find_file $srcdir/$subdir/$1.tar     "tar xf"     && return
            find_file $srcdir/$subdir/$1.tar.bz2 "tar xfj"    && return
            find_file $srcdir/$subdir/$1.tgz     "tar xfz"    && return
            find_file $srcdir/$subdir/$1.zip     "unzip -q"   && return
            find_file $srcdir/$subdir/$1.jar     "jar xf"     && return
```

```
            done
        done
    }
```

很明显：内部循环里的 find_package 是在识别多种存档文件格式，而另一个函数 find_file则是在执行真正的操作时被调用，当它成功时，可立即返回。内部循环的第二次迭代中，subdir 为空，且路径名称里也有两个连续的斜杠，不过这没关系，我们在附录B中有说明。虽然这个程序代码乍看上去与例8-1里的 pathfind 命令类似，然而，这里我们需要在每个目录下寻找许多文件，且针对它们做点不一样的处理。

我们在本节一开始提过：.tar.gz 的存档文件格式是很普遍的。不过当然还是可能出现其他压缩与命名结构（scheme）。tar 是 UNIX 主要的命令，尽管其他操作系统里也有 tar 的实作，但它们都不包含在标准发布中。InfoZip 格式（注 3）为许多人协同开发的产物，目标是支持任何操作系统上所使用的压缩存档文件，而 Java 的 jar（注 4）文件使用的就是 InfoZip 格式。在 find_package 里的循环内容可完全处理这些文件。

以小型站点来看，将包存档文件存储在单一目录下或许是合理的，例如：/usr/local/src。然而，当存档文件不断增大时，这样的组织方式就会变得很笨重。在我们的站点里，每个包都被给定它自己的来源目录，例如 gawk 3.1.4 版的存档文件，我们就将它放在 /usr/local/gnu/src/gawk/gawk-3.1.4.tar.gz，而该版本的构建日志则存储于 /usr/local/gnu/src/gawk/logs/gawk-3.1.4。每个包目录内的WHERE-FROM文件都记录了包在Internet上的主要存档文件位置，以便我们检查较新的版次。一般来说，我们会保留包存档文件最新的数版，因为有一天可能在网络无法运行，或远程主要存档文件站点无法连接时，这时当你需要重建包时，就派得上用场了。因此，find_package 里的循环体会从包名称中将版本编号截断，存储结果到 base 中，并在退回去查阅 $srcdir 之前，它会先在 $srcdir/$base 中查找包。

我们发现保留构建日志相当有用，因为在安装一段时日后，当你在探究 bug 而需要有关使用哪个编译器与要套用哪些选项等细节时，一定会用得到它。而且，对一些可移植性较差的包来说，经常会需要在构建程序中，或甚至是针对来源文件做些小调整，以利构建完成。如果该信息都记录在日志文件里，则在安装那些包的较新版本时便能节省一些时间了。

find_file函数是用来测试包存档文件的可读性以及存在与否，再将其参数记录在两个全局变量里，最后返回状态结果。这样大大地简化了 find_package 程序代码：

注 3： 见 *http://www.info-zip.org/*。

注 4： jar 文件可包含校验和与数字签名，可用以检测文件是否有误或遭篡改，所以它们出现在一般的软件分发中是越来越频繁了。

```
find_file()
{
    # Usage:
    #       find_file file program-and-args
    # Return 0 (success) if found, 1 (failure) if not found

    if test -r "$1"
    then
        PAR="$2"                          用来提取的程序与参数
        PARFILE="$1"                      要提取来源的实际文件
        return 0
    else
        return 1
    fi
}
```

set_userhosts函数允许用户指定确切的路径（可能是与目前目录相对的路径），也可以是在$BUILDHOME初始化目录中所找到的，来提供userhosts文件。对于已知只能运行在某种限定环境下的包来说，这么做便于将构建主机群按照编译器、平台或包加以分组。可以提供任何数目的userhosts文件，所以我们只要简单地将它们的名称累积在ALTUSERHOSTS里就可以了：

```
set_userhosts()
{
    # Usage: set_userhosts file(s)
    for u in "$@"
    do
        if test -r "$u"
        then
            ALTUSERHOSTS="$ALTUSERHOSTS $u"
        elif test -r "$BUILDHOME/$u"
        then
            ALTUSERHOSTS="$ALTUSERHOSTS $BUILDHOME/$u"
        else
            error "File not found: $u"
        fi
    done
}
```

最后一个函数build_one，便是处理远程主机上的包作业。因为函数太长，我们拆为几个部分讲解：

```
build_one()
{
    # Usage:
    #       build_one [user@]host[:build-directory][,envfile]
```

现在，除了在注释标志中有简短提及外，我们并未精确地指出$HOME/.build/userhosts初始文件里到底是什么数据。这里我们得切分成4段信息：远程主机上的用户名称（与初始化主机上的不相同时）、主机名称本身、应该构建的远程主机上已存在

的目录名称，以及构建时可能的特定额外环境变量的设置。在Shell脚本里，将这些信息分别存储在独立的文件里会很不方便，所以我们借用远程与secure Shell的语法，将它们整合在一起，并以分隔字符隔开，像这样：

```
jones@freebsd.example.com:/local/build,$HOME/.build/c99
```

仅主机名称部分是强制性的。

我们也会需要用到这些部分，所以这里使用echo与sed将参数分割。通过eval传递参数，并展开名称中的任何环境变量（例如$HOME/.build/c99里的HOME），以避免在userhosts文件里，将系统特定的登录目录路径直接编码。为方便起见，如果没有指定，则我们会提供/tmp作为默认的构建目录：

```
arg="`eval echo $1`"                                    展开环境变量

userhost="`echo $arg | sed -e 's/:.*$//'`"              删除冒号与冒号后的
                                                        任何东西

user="`echo $userhost | sed -e s'/@.*$//'`"             取出用户名称
test "$user" = "$userhost" && user=$USER                如为空，则使用$USER

host="`echo $userhost | sed -e s'/^[^@]*@//'`"          取出主机信息

envfile="`echo $arg | sed -e 's/^[^,]*,//'`"            环境变量文件名称
test "$envfile" = "$arg" && envfile=/dev/null

builddir="`echo $arg | sed -e s'/^.*://' -e 's/,.*//'`" 构建目录
test "$builddir" = "$arg" && builddir=/tmp
```

我们希望能找一个较固定（安定）的临时目录给予builddir使用，但不同的UNIX厂商之间，临时目录的名称并不一致。虽然几行额外程序便能做些测试，不过我们还是假定大部分的用户会指定一个合理的构建目录。除了/tmp常会在重开机后被清除内容的原因，另外还有一些理由让我们认为/tmp不是builddir好的选择：

- 在许多系统上，/tmp是一个分别的文件系统，它可能太小以至于无法处理庞大的包构建树状结构。
- 在部分系统上，/tmp是以不具备执行程序权限的方式加载的：这可能导致configure的测试与验证检查失败。
- 在某些Sun Solaris的版本下，由于不明的因素，无法让本地编译器编译/tmp下的程序代码。

envfile工具是相当重要的：它让我们能覆盖掉在configure里已作的默认选择。软件开发人员当然应尽可能地测试各种编译器，以验证软件的可移植性并调试。通过选择不同的构建目录与envfile的值，我们可以在同一台主机上，以不同的编译器同时执行多个构建。envfile文件十分简单，它们只是设置环境变量而已，如下所示：

```
$ cat $HOME/.build/c99
CC=c99
CXX=CC
```

程序的下一步：将仅含文件名（bare filename）的部分，例：gawk-3.1.4.tar.gz，存储到parbase变量中：

```
parbase=`basename $PARFILE`
```

包名称（例：gawk-3.1.4）则存储到变量package：

```
package="`echo $parbase | \
        sed     -e 's/[.]jar$//' \
                -e 's/[.]tar[.]bz2$//' \
                -e 's/[.]tar[.]gz$//' \
                -e 's/[.]tar[.]Z$//' \
                -e 's/[.]tar$//' \
                -e 's/[.]tgz$//' \
                -e 's/[.]zip$//'`"
```

我们使用显式的sed模式切去字尾，因为在名称中有太多的点号，较简化的模式更可信赖。为确保它们也可以与旧式的sed实现一起运行，我们是以分开的替换命令而不是以单一扩展正则表达式指明它们。如需支持已加入到find_package的新存档文件格式，则也应在此处更新这些编辑器模式。

下一步便是将存档文件复制到远程主机上的构建目录，除非它已经出现在该系统上，可能是通过加载文件系统或是映射的方式。这种做法在我们的站点很常见，所以这个检查操作可节省时间与磁盘空间。

虽然我们通常会避免编写成聊天程序，不过在每次与远程系统通信之前先执行echo命令其实是相同的：它让用户得到必要的反馈信息。远程复制是很耗时的，且很可能会出现失败或停滞不动：没有这样的反馈信息，用户很难了解到底为什么脚本执行了那么久，或究竟是哪台主机导致错误的发生。parbaselocal变量则是用来区分出存档文件的临时版本与先前已存在之版本的差别：

```
echo $SSH $SSHFLAGS $userhost "test -f $PARFILE"
if $SSH $SSHFLAGS $userhost "test -f $PARFILE"
then
    parbaselocal=$PARFILE
else
    parbaselocal=$parbase
    echo $SCP $PARFILE $userhost:$builddir
    $SCP $PARFILE $userhost:$builddir
fi
```

理想上，使用管道解包比较好，因为这么做可以将输入/输出的量减半，另外磁盘空间需求也会减半。可惜的是，只有jar和tar可以用该方式读取它们的存档文件：unzip

是需要实际文件的。实际上，jar 可以读取 InfoZip 文件，这让我们可以用 jar 置换 unzip，并使用管道。遗憾的是，编写本书的时候，jar 仍不够成熟，我们至少就发现有一个实现会卡在 .zip 文件的处理上。

远程复制是连续执行，而非并行处理。后者不是不可能做到，只是会增加主要程序额外的复杂性，它必须先寻找与分发包，等待包分发完成，接着再构建。当然，这么一来构建时间就会比远程复制时间长很多，所以连续性复制不会占去太多的总执行时间。

我们的日志文件以包、远程主机，及以秒计的时间戳命名。如果在单一远程主机上执行多个构建，则可能会有文件名冲突的风险。在日志文件名内使用进程编号变量，$$，也不是个好的解决方案，因为它在build-all单一调用内是固定常数。我们虽然可以使用$$初始化计数器，这么做会在每次构建时增值且被用在日志文件的文件名中，但结果只会让无意义的数字弄乱文件名。解决方案是在两个连续日志文件的产生之间，至少具有一秒的间隔：sleep 正是我们所需的。GNU 的 date 提供 %N（nanoseconds 十亿分之一秒）格式项，应该可以满足产生唯一的文件名的需求，无须用到 sleep，但 POSIX 与旧式 date 实现，缺乏这个格式项。为满足最大的可移植性，我们就由秒数处理：

```
sleep 1
now="`date $DATEFLAGS`"
logfile="$package.$host.$now.log"
```

接下来要进入这个说明的最后部分：用于远程主机上实现构建的长命令。$SSH 前置 nice命令是为了降低它的优先权，以避免与系统上的交互式工作竞争资源。即便很多工作都是在远程系统上做的，但构建日志有时会很大，让 $SSH 要作的事情变多。

留意 $SSH 的第二个参数是以双引号界定的长字符串。在该字符串中，以货币符号前置的变量将“于脚本的内文中被展开”，且在远程主机上无须被知道。

我们在 $SSH 参数字符串里所需要用到的命令语法根据远程主机上用户的登录 Shell 而定。我们极小心地限制该语法，让它能在所有一般性 UNIX Shell 内正常运行，这么一来才能让所有用户，即便是在不同主机上、使用不同登录Shell者都能使用build-all。我们无法要求任何地方都使用相同的登录Shell，因为在很多系统上，用户无法选择自己要用的Shell。替代方案就是使用管道，将命令流传给每个主机上的Bourne Shell，不过这么做会为每一个构建启动另一个进程，而让我们陷入更深的混乱：一次处理三个Shell 已经够难的了。

```
nice $SSH $SSHFLAGS $userhost "
    echo '============================================================' ;
```

在登录Shell先出现在命令序列的情况下，如果 $BUILDBEGIN脚本存在，则在远程系统上执行。这么做可提供登录定制，例如当Shell启动文件无法增加PATH变量时（例ksh

与sh)。它也会将一些额外的信息写至标准错误输出或标准输出，这样当然也可以写到构建日志文件中。Bourne-Shell家族里的Shell使用点号命令来执行目前目录中的命令，而C-Shell家族里的Shell则使用source命令。bash与zsh Shell支持这两种命令。

问题是，如果点号命令指定的文件不存在，则有些Shell，包括POSIX的，会中止执行点号命令。这么一来，尽管true命令是用于条件式的结尾处，也会使得单纯的 . $BUILDBEGIN || true程序代码失败。因此，我们还会需要文件存在与否的测试，而且也必须处理source命令。因为两个Shell都认得点号命令与source命令两者，所以我必须在单一的复杂命令里做这件事，这条命令依赖于布尔运算符相等的优先权：

```
test -f $BUILDBEGIN && . $BUILDBEGIN || \
    test -f $BUILDBEGIN && source $BUILDBEGIN || \
        true ;
```

我们不爱用这么复杂的语句，但build-all严格的设计需求，务求可运行于所有登录Shell中，让我们不得不这么做，我们也找不到更简化的可接受解决方案。

我们假设在build-all使用之前，已做过启动脚本的调试了。否则，如果$BUILDBEGIN脚本的执行是在错误状态下终止的话，则它可能会被试图执行两次。

长久以来的使用经验告诉我们，记录在构建日志里的额外信息会很有用，所以下面一连串的echo命令就是为了这个目的，特意安排的格式只是为了让日志文件更容易阅读：

```
echo 'Package:                 $package' ;
echo 'Archive:                 $PARFILE' ;
echo 'Date:                    $now' ;
echo 'Local user:              $USER' ;
echo 'Local host:              `hostname`' ;
echo 'Local log directory:     $LOGDIR' ;
echo 'Local log file:          $logfile' ;
echo 'Remote user:             $user' ;
echo 'Remote host:             $host' ;
echo 'Remote directory:        $builddir' ;
```

有时知道花了多少时间构建也是很有用的（我们有一台较老旧的系统，在构建GNU C编译器时花了将近一天），所以我们的脚本也会报告开始与结束的日期。这些全取自于远程主机，因为每台主机的时区有可能不同，也可能会有时间差的问题，而且稍后将已安装文件的时间戳与构建日志的项目作匹配，也是很重要的。由于echo没有适用的可移植式用法，所以我们使用printf：

```
printf 'Remote date:
date $DATEFLAGS ;
```

同样地，我们也记录系统与GNU编译器的版本信息，因为在日后的错误报告里会需要用到：

```
printf 'Remote uname:          ' ;
uname -a || true ;
printf 'Remote gcc version:    ' ;
gcc --version | head -n 1 || echo ;
printf 'Remote g++ version:    ' ;
g++ --version | head -n 1 || echo ;
```

由于其他编译器没有一致的方式可取得版本信息，所以我们无法在build-all中处理该工作，取而代之的是，我们可以自$BUILDBEGIN脚本里，通过适当的命令，产生想要的报告。接下来，我们的脚本提供其他信息如下：

```
echo 'Configure environment:  `$STRIPCOMMENTS $envfile | $JOINLINES`' ;
echo 'Extra environment:      $EXTRAENVIRONMENT' ;
echo 'Configure directory:    $CONFIGUREDIR' ;
echo 'Configure flags:        $CONFIGUREFLAGS' ;
echo 'Make all targets:       $ALLTARGETS' ;
echo 'Make check targets:     $CHECKTARGETS' ;
```

磁盘空间耗尽是常见的导致错误发生的原因，所以我们在构建的前与后，都使用df报告可用空间：

```
echo 'Disk free report for $builddir/$package:' ;
df $builddir | $INDENT ;
```

configure与make都可能被环境变量影响，所以我们最后的一道工作就是排序日志文件标头：

```
echo 'Environment:' ;
env | env LC_ALL=C sort | $INDENT ;
echo '=============================================================' ;
```

管道中间的env命令乃是为了确保脚本可正常运行于所有Shell下，包括C-Shell家族成员。

和在本地的设置相同，我们也在远程系统上设置权限掩码，允许组成员完整访问，并且除此之外的其他人都有读取权限：

```
umask $UMASK ;
```

包存档文件已经存在于构建目录内，所以我们切换到该目录，如果cd失败，则以错误状态退出：

```
cd $builddir || exit 1 ;
```

下一步：删除所有旧的存档文件树。这里使用rm的绝对路径，因为这些命令执行于Shell交互模式下，而有些站点对于这个命令将其设置别名为具有 -i 的交互式选项：

```
/bin/rm -rf $builddir/$package ;
```

有时我们会因为改变了编译器或编译选项，而要再执行一次构建操作，所以递归删除是有其必要的，这可以确保我们能够从干净的分发开始。rm里的-f选项是要求静默地处理对于不存在目录的任何抱怨。

一个递归文件树的删除是相当危险的操作，也可能成为攻击的目标。因为package是从信赖的basename命令中取得，我们能够确信它不包含斜杠，因此可以只参照到当前目录。将$builddir/加入到rm的参数中，则可提供起码的安全性，不过还是不够，因为不管是builddir或者是package都可能被设置为一个点号，即当前的目录。

这种情况确实会沦为安全漏洞，而我们也无法再多作些什么保护，所能做的只有警告标语。很明显，这个程序应该绝对不要以root身份执行。我们在脚本启动的一开始，就使用此语句，则可阻止用户这么做：

```
test "`id -u`" -eq 0 && \
    error For security reasons, this program must NOT be run by root
```

在我们的所有系统里，只有Sun Solaris的id缺乏-u选项的支持，不过我们设置了PATH，让程序先找到GNU *coreutil*版本的id。

注意： 包安装命令常告诉你：请以root账号构建与安装软件，其实你应该要忽略此指示：因为只有极少数的包需要这样的权限，而且，即便是要，也只有在安装步骤才需要。

接下来，解开存档文件：

```
$PAR $parbaselocal ;
```

重要的是你必须了解：$PAR是在初始化主机上被展开，但在远程主机上执行。特别是我们假设tar是支持-j与-z选项的GNU版本，且unzip与jar都可用。对于这个脚本的每个用户，我们都预期他已在每个远程主机上做好Shell启动文件的适当设置，确保这些程序都能被找到。我们不能为这些程序提供固定的路径，因为这些路径在每个远程主机上都可能不尽相同。

如果存档文件已复制到远程主机上，则parbaselocal与parbase会是一致的值，且因为远程主机已不再需要包存档文件，所以我们将它删除：

```
test "$parbase" = "$parbaselocal" && /bin/rm -f $parbase ;
```

我们已准备好切换到包目录开始构建。对于遵循广泛使用的GNU Project惯例的软件包来说：该目录为包目录的最顶层。遗憾的是，有些包会将构建目录埋在文件树的较深处，像用来编写脚本以及加速构建窗口系统界面的Tcl与Tk工具就是。命令行的--cd选项

提供了存储在CONFIGUREDIR里的构建目录的相对路径，用来覆盖掉它的点号（.）默认值。我们接下来会需要用到package与CONFIGUREDIR这两个变量以切换至构建目录，且如果切换失败，会以错误码退出：

```
cd $package/$CONFIGUREDIR || exit 1 ;
```

许多包现已含有configure脚本，所以我们可以试试它，如果找到，则以envfile所提供的任何额外环境变量执行它。我们也会传递任何由--configure选项提供的额外标志。大部分包不需要这类标志，不过有些较复杂的就要：

```
test -f configure && \
    chmod a+x configure && \
        env `$STRIPCOMMENTS $envfile | $JOINLINES` \
            $EXTRAENVIRONMENT \
                nice time ./configure $CONFIGUREFLAGS ;
```

chmod命令乃用来加入执行权限，在这里使用的理由有两个：首先，由于我们偶尔会遇到缺乏该权限的包存档文件，再则，是因为现行Java的jar存档文件格式会忽略记录该权限（注5）。前置nice命令可降低工作优先权，这么做可对远程系统的影响降到最低。前置time命令，可报告configure执行时间。我们见过一些非常大的配置脚本，这么做有助于记录它们的执行时间以作为下一版构建时间的估计。

我们现在要进入操作最多的地方：实际的构建与包的验证，一样是前置nice time，并由--all与--check选项（或它们的默认值）提供的make参数：

```
nice time make $ALLTARGETS && nice time make $CHECKTARGETS ;
```

make命令背后隐藏了很多工作，不过完成该工作的规则已通过开发人员写在Makefile里了，我们终端安装人员通常不必理会内容是什么。

我们希望成功构建完成时，在日志文件内看到报告类似All tests passed!这样的信息，或其他容易理解的报告，让我们知道一切都好。验证测试是非常重要，绝不应该跳过。即便是包在开发站点里已运行无误，但没有理由相信它在我们的站点里也会运行得这么顺利，因为有这么多的东西可能导致它出错：不同的系统架构、编译器、环境变量、文件系统、本地定制设置与调校、操作系统版本、查找路径、共享函数库、系统标头文件、X Windows System默认值等等，很多都有可能导致错误发生。

我们现已包装好远程命令，还伴随几行日志文件中最后的报告：

```
echo '=================================================================' ;
echo 'Disk free report for $builddir/$package:' ;
```

注5：　这很有可能是设计上的瑕疵，因为底层的InfoZip格式支持它。

```
df $builddir | $INDENT ;
printf 'Remote date:              ' ;
date $DATEFLAGS ;
```

$BUILDEND脚本就像$BUILDBEGIN脚本一样，在根目录下，提供任何最后的额外日志文件报告，但true是确保成功地完成：

```
cd ;
test -f $BUILDEND && . $BUILDEND || \
    test -f $BUILDEND && source $BUILDEND || \
        true ;
echo '==========================================================' ;
```

build_one函数的最后两行是关闭远程命令列表与函数体，重定向标准输出与标准错误输出两者到日志文件。最重要的是，在后台执行远程命令，使得该执行可以马上在主体的内部循环中继续。远程Shell的输入被重定向到null设备，所以它不会悬在那里等待用户输入：

```
    " < /dev/null > "$LOGDIR/$logfile" 2>&1 &
}
```

这样大小的程序与其功能，必定需要在线帮助。由于篇幅的关系，不允许我们在此展现build-all的手册页，不过此脚本与它的手册页文件都在本书网站上。

完整的脚本具有一些注释，且在开始处以字母顺序重新排序函数，这些都整理在例8-2里。虽然这有320行之多（省略注释与空行的情况下），但花时间了解我们写程序的方式，其实是很受用的。一旦新分发的包被取到本地系统上时，一个单行命令便能以并行处理的方式，在所有构建主机上启动构建与验证。经过一段时间的等待之后，安装程序会检查构建日志以得知它们的成功或失败，并决定在哪些主机上可以安全地执行make install。将软件安装在系统上，之后构建目录便可以从远程系统上删除了。

注意： 构建失败不能归因于本地的错误时，则应报告给包开发人员。很少有开发人员会广泛使用各种平台，所以唯有来自安装者的反馈，他们才能作出更具可移植性且健全的包。在执行之前，你当然应该先看看包的发行注意事项（多半是在叫做BUGS、FAQ、INSTALL、PROBLEMS或README的文件内），再看看是不是你发现的这个问题已经有人提出了，只是尚未修正。在这样的软件模式下，开发人员可以很快地得到安装者的反馈，最后的结果就是高生产力，而Eric Raymond也将此编写成书（注6）。

注6：《The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary》(O'Reilly).

例 8-2：build-all 程序

```
#! /bin/sh -
# 在一台或多台构建主机上，并行构建一个或多个包
#
# 语法：
#       build-all [ --? ]
#                 [ --all "..." ]
#                 [ --check "..." ]
#                 [ --configure "..." ]
#                 [ --environment "..." ]
#                 [ --help ]
#                 [ --logdirectory dir ]
#                 [ --on "[user@]host[:dir][,envfile] ..." ]
#                 [ --source "dir ..." ]
#                 [ --userhosts "file(s)" ]
#                 [ --version ]
#                 package(s)
#
# 可选用的初始化文件：
#       $HOME/.build/directories        list of source directories
#       $HOME/.build/userhosts          list of [user@]host[:dir][,envfile]

IFS='
        '

PATH=/usr/local/bin:/bin:/usr/bin
export PATH

UMASK=002
umask $UMASK

build_one()
{
    # 语法：
    #       build_one [user@]host[:build-directory][,envfile]
    arg="`eval echo $1`"

    userhost="`echo $arg | sed -e 's/:.*$//'`"

    user="`echo $userhost | sed -e s'/@.*$//'`"
    test "$user" = "$userhost" && user=$USER

    host="`echo $userhost | sed -e s'/^[^@]*@//'`"

    envfile="`echo $arg | sed -e 's/^[^,]*,//'`"
    test "$envfile" = "$arg" && envfile=/dev/null

    builddir="`echo $arg | sed -e s'/^.*://' -e 's/,.*//'`"
    test "$builddir" = "$arg" && builddir=/tmp

    parbase=`basename $PARFILE`

    # NB: 如果这些模式被更换过，则更新 find_package()
       package="`echo $parbase | \
```

```
        sed     -e 's/[.]jar$//' \
                -e 's/[.]tar[.]bz2$//' \
                -e 's/[.]tar[.]gz$//' \
                -e 's/[.]tar[.]Z$//' \
                -e 's/[.]tar$//' \
                -e 's/[.]tgz$//' \
                -e 's/[.]zip$//'`"

# 如果我们在远程主机上看不到包文件，则复制过去
echo $SSH $SSHFLAGS $userhost "test -f $PARFILE"
if $SSH $SSHFLAGS $userhost "test -f $PARFILE"
then
    parbaselocal=$PARFILE
else
    parbaselocal=$parbase
    echo $SCP $PARFILE $userhost:$builddir
    $SCP $PARFILE $userhost:$builddir
fi

# 在远程主机上解开存档文件、构建，
# 及以后台执行方式检查它

sleep 1         # 为了保证唯一的日志文件名
now="`date $DATEFLAGS`"
logfile="$package.$host.$now.log"
nice $SSH $SSHFLAGS $userhost "
    echo '==============================================================' ;
    test -f $BUILDBEGIN && . $BUILDBEGIN || \
        test -f $BUILDBEGIN && source $BUILDBEGIN || \
            true ;
    echo 'Package:                 $package' ;
    echo 'Archive:                 $PARFILE' ;
    echo 'Date:                    $now' ;
    echo 'Local user:              $USER' ;
    echo 'Local host:              `hostname`' ;
    echo 'Local log directory:     $LOGDIR' ;
    echo 'Local log file:          $logfile' ;
    echo 'Remote user:             $user' ;
    echo 'Remote host:             $host' ;
    echo 'Remote directory:        $builddir' ;
    printf 'Remote date:             ' ;
    date $DATEFLAGS ;
    printf 'Remote uname:            ' ;
    uname -a || true ;
    printf 'Remote gcc version:      ' ;
    gcc --version | head -n 1 || echo ;
    printf 'Remote g++ version:      ' ;
    g++ --version | head -n 1 || echo ;
    echo 'Configure environment:   `$STRIPCOMMENTS $envfile | $JOINLINES`' ;
    echo 'Extra environment:       $EXTRAENVIRONMENT' ;
    echo 'Configure directory:     $CONFIGUREDIR' ;
    echo 'Configure flags:         $CONFIGUREFLAGS' ;
    echo 'Make all targets:        $ALLTARGETS' ;
    echo 'Make check targets:      $CHECKTARGETS' ;
```

```
        echo 'Disk free report for    $builddir/$package:' ;
        df $builddir | $INDENT ;
        echo 'Environment:' ;
        env | env LC_ALL=C sort | $INDENT ;
        echo '==============================================================' ;
        umask $UMASK ;
        cd $builddir || exit 1 ;
        /bin/rm -rf $builddir/$package ;
        $PAR $parbaselocal ;
        test "$parbase" = "$parbaselocal" && /bin/rm -f $parbase ;
        cd $package/$CONFIGUREDIR || exit 1 ;
        test -f configure && \
            chmod a+x configure && \
                env `$STRIPCOMMENTS $envfile | $JOINLINES` \
                    $EXTRAENVIRONMENT \
                        nice time ./configure $CONFIGUREFLAGS ;
        nice time make $ALLTARGETS && nice time make $CHECKTARGETS ;
        echo '==============================================================' ;
        echo 'Disk free report for $builddir/$package:' ;
        df $builddir | $INDENT ;
        printf 'Remote date:            ' ;
        date $DATEFLAGS ;
        cd ;
        test -f $BUILDEND && . $BUILDEND || \
            test -f $BUILDEND && source $BUILDEND || \
                true ;
        echo '==============================================================' ;
    " < /dev/null > "$LOGDIR/$logfile" 2>&1 &
}

error()
{
    echo "$@" 1>&2
    usage_and_exit 1
}

find_file()
{
    # 语法:
    #       find_file file program-and-args
    # 如果找到，返回 0 (成功)，如果找不到则返回 1 (失败)
    if test -r "$1"
    then
        PAR="$2"
        PARFILE="$1"
        return 0
    else
        return 1
    fi
}

find_package()
{
    # 语法：find_package package-x.y.z
```

```
    base=`echo "$1" | sed -e 's/[-_][.]*[0-9].*$//'`
    PAR=
    PARFILE=
    for srcdir in $SRCDIRS
    do
        test "$srcdir" = "." && srcdir="`pwd`"

        for subdir in "$base" ""
        do
            # NB: 如果此列表有改变，则更新 build_one() 内的包设置
            find_file $srcdir/$subdir/$1.tar.gz  "tar xfz"    && return
            find_file $srcdir/$subdir/$1.tar.Z   "tar xfz"    && return
            find_file $srcdir/$subdir/$1.tar     "tar xf"     && return
            find_file $srcdir/$subdir/$1.tar.bz2 "tar xfj"    && return
            find_file $srcdir/$subdir/$1.tgz     "tar xfz"    && return
            find_file $srcdir/$subdir/$1.zip     "unzip -q"   && return
            find_file $srcdir/$subdir/$1.jar     "jar xf"     && return
        done
    done
}

set_userhosts()
{
    # 语法: set_userhosts file(s)
    for u in "$@"
    do
        if test -r "$u"
      then
            ALTUSERHOSTS="$ALTUSERHOSTS $u"
        elif test -r "$BUILDHOME/$u"
        then
            ALTUSERHOSTS="$ALTUSERHOSTS $BUILDHOME/$u"
        else
            error "File not found: $u"
        fi
    done
}

usage()
{
    cat <<EOF
Usage:
        $PROGRAM [ --? ]
                        [ --all "..." ]
                        [ --check "..." ]
                        [ --configure "..." ]
                        [ --environment "..." ]
                        [ --help ]
                        [ --logdirectory dir ]
                        [ --on "[user@]host[:dir][,envfile] ..." ]
                        [ --source "dir ..." ]
                        [ --userhosts "file(s)" ]
                        [ --version ]
                        package(s)
```

```
EOF
}

usage_and_exit()
{
    usage
    exit $1
}

version()
{
    echo "$PROGRAM version $VERSION"
}

warning()
{
    echo "$@" 1>&2
    EXITCODE=`expr $EXITCODE + 1`
}

ALLTARGETS=
altlogdir=
altsrcdirs=
ALTUSERHOSTS=
BUILDBEGIN=./.build/begin
BUILDEND=./.build/end
BUILDHOME=$HOME/.build
CHECKTARGETS=check
CONFIGUREDIR=.
CONFIGUREFLAGS=
DATEFLAGS="+%Y.%m.%d.%H.%M.%S"
EXITCODE=0
EXTRAENVIRONMENT=
INDENT="awk '{ print \"\t\t\t\" \$0 }'"
JOINLINES="tr '\n' '\040'"
LOGDIR=
PROGRAM=`basename $0`
SCP=scp
SSH=ssh
SSHFLAGS=${SSHFLAGS--x}
STRIPCOMMENTS='sed -e s/#.*$//'
userhosts=
VERSION=1.0

# 默认的初始化文件
defaultdirectories=$BUILDHOME/directories
defaultuserhosts=$BUILDHOME/userhosts

# 要寻找包分发的位置列表，
# 如果用户未提供个人化列表，则使用默认列表：
SRCDIRS="`$STRIPCOMMENTS $defaultdirectories 2> /dev/null`"
test -z "$SRCDIRS" && \
    SRCDIRS="
```

```
            /usr/local/src
            /usr/local/gnu/src
            $HOME/src
            $HOME/gnu/src
            /tmp
            /usr/tmp
            /var/tmp
        "

while test $# -gt 0
do
    case $1 in

    --all | --al | --a | -all | -al | -a )
        shift
        ALLTARGETS="$1"
        ;;

    --cd | -cd )
        shift
        CONFIGUREDIR="$1"
        ;;

    --check | --chec | --che | --ch | -check | -chec | -che | -ch )
        shift
        CHECKTARGETS="$1"
        ;;

    --configure | --configur | --configu | --config | --confi | \
    --conf | --con | --co | \
    -configure | -configur | -configu | -config | -confi | \
    -conf | -con | -co )
        shift
        CONFIGUREFLAGS="$1"
        ;;

    --environment | --environmen | --environme | --environm | --environ | \
    --enviro | --envir | --envi | --env | --en | --e | \
    -environment | -environmen | -environme | -environm | -environ | \
    -enviro | -envir | -envi | -env | -en | -e )
        shift
        EXTRAENVIRONMENT="$1"
        ;;

    --help | --hel | --he | --h | '--?' | -help | -hel | -he | -h | '-?' )
        usage_and_exit 0
        ;;

    --logdirectory | --logdirector | --logdirecto | --logdirect | \
    --logdirec | --logdire | --logdir | --logdi | --logd | --log | \
    --lo | --l | \
    -logdirectory | -logdirector | -logdirecto | -logdirect | \
    -logdirec | -logdire | -logdir | -logdi | -logd | -log | -lo | -l )
        shift
        altlogdir="$1"
        ;;
```

```
    --on | --o | -on | -o )
        shift
        userhosts="$userhosts $1"
        ;;

    --source | --sourc | --sour | --sou | --so | --s | \
    -source | -sourc | -sour | -sou | -so | -s )
        shift
        altsrcdirs="$altsrcdirs $1"
        ;;

    --userhosts | --userhost | --userhos | --userho | --userh | \
    --user | --use | --us | --u | \
    -userhosts | -userhost | -userhos | -userho | -userh | \
    -user | -use | -us | -u )
        shift
        set_userhosts $1
        ;;

    --version | --versio | --versi | --vers | --ver | --ve | --v | \
    -version | -versio | -versi | -vers | -ver | -ve | -v )
        version
        exit 0
        ;;

    -*)
        error "Unrecognized option: $1"
        ;;

    *)
        break
        ;;

    esac
    shift
done

# 寻找适当的邮件客户端程序
for MAIL in /bin/mailx /usr/bin/mailx /usr/sbin/mailx /usr/ucb/mailx \
            /bin/mail /usr/bin/mail
do
    test -x $MAIL && break
done
test -x $MAIL || error "Cannot find mail client"

# 命令行来源目录优先于默认值
SRCDIRS="$altsrcdirs $SRCDIRS"

if      test -n "$userhosts"
then
    test -n "$ALTUSERHOSTS" &&
        userhosts="$userhosts `$STRIPCOMMENTS $ALTUSERHOSTS 2> /dev/null`"
else
    test -z "$ALTUSERHOSTS" && ALTUSERHOSTS="$defaultuserhosts"
    userhosts="`$STRIPCOMMENTS $ALTUSERHOSTS 2> /dev/null`"
fi
```

```
# 检查是否要执行某些操作
test -z "$userhosts" && usage_and_exit 1

for p in "$@"
do
    find_package "$p"

    if test -z "$PARFILE"
    then
        warning "Cannot find package file $p"
        continue
    fi

    LOGDIR="$altlogdir"
    if test -z "$LOGDIR" -o ! -d "$LOGDIR" -o ! -w "$LOGDIR"
    then
        for LOGDIR in "`dirname $PARFILE`/logs/$p" $BUILDHOME/logs/$p \
                      /usr/tmp /var/tmp /tmp
        do
            test -d "$LOGDIR" || mkdir -p "$LOGDIR" 2> /dev/null
            test -d "$LOGDIR" -a -w "$LOGDIR" && break
        done
    fi

    msg="Check build logs for $p in `hostname`:$LOGDIR"
    echo "$msg"
    echo "$msg" | $MAIL -s "$msg" $USER 2> /dev/null

    for u in $userhosts
    do
        build_one $u
    done
done
# 将退出状态限制为一般 UNIX 实际的做法
test $EXITCODE -gt 125 && EXITCODE=125

exit $EXITCODE
```

8.3 小结

在本章中，我们写了 UNIX 系统里现在还没有的两个好用工具，使用 Shell 语句与现有的标准工具完成任务。不管是它们的哪一个，执行时都不会特别耗时，所以用户应该不会想以程序语言 C 或 C++ 将它们重写。以 Shell 脚本来说，它们可以完全不做任何更改，即可在大部分现代 UNIX 平台上执行。

这两个程序都支持命令行选项，这些选项可干净地被 `while` 与 `case` 语句处理。两者都使用 Shell 函数以简化处理且避免不必要的程序代码重复。最后，这些程序也在安全性议题上花了相当的心思，并对它们的参数与变量执行了健康检查。

第9章

awk的惊人表现

awk程序语言的设计，就是为了简化一般文本处理的工作。在本章中，我们将介绍Shell脚本里有关awk的部分。

关于更高级的awk语言处理，你可以阅读引用书目里的图书。如果你的系统里是安装GNU的gawk，也可以在在线info系统（注1）里找到它的使用手册。

所有UNIX系统里都至少有一套awk。该语言在20世纪80年代中期大举扩张版图，部分厂商仍维持旧的awk实现，且有时称为oawk，之后新产品则取名为nawk。IBM AIX与Sun Solaris都延续这样的实现方式，不过除此之外的其他系统当前仅提供新版。Solaris下的POSIX兼容版本放在/usr/xpg4/bin/awk。在本书中，我们仅考虑已扩展的语言并称之为awk——无论你系统里必须使用的是nawk、gawk或是mawk。

先承认我们对awk有强烈偏见：因为太喜欢它了。我们实现它、维护它、移植它，并用它编写程序多年。即便短小精悍的awk程序很多，但我们有些大型的awk程序是上千行的。awk的简单与强大功能，使其看来就像是为了某个工作而设计的工具，我们在awk上很少遇到需要某种文本处理工作却找不到可用的功能或者很难实现的情况。我们曾试着以C或C++重写一个awk程序，结果是程序更长、很难调试，而且执行的速度也只不过稍快一些而已。

不同于其他脚本语言的是awk拥有多个实现，这种健康的情况鼓励用户拥护一种通用的语言，同时也允许用户在这之间自由地切换使用。再者，awk是POSIX的一部分，并拥有非UNIX操作系统的实现，这也是它不同于其他脚本语言之处。

注1： GNU文件查看程序：info是*texinfo*包的一部分，可从*ftp://ftp.gnu.org/gnu/textinfo/*中获得。emacs文字编辑器也可用于访问该文件：在emacs的session中，按Ctrl-H即可。

如果你的本机系统上的awk不是标准版本，你可以引用表9-1，取得其中任一个免费的实现。这些程序全都具有充分的可移植性且易于安装。gawk可作为类似实验台的功能，用来提供好玩的新内建函数及语言功能，包括网络I/O、性能探测、国际化以及可移植性检查。

表9-1：供免费取用的awk版本

程序	位置
贝尔实验室的awk	*http://cm.bell-labs.com/who/bwk/awk.tar.gz*
gawk	*ftp://ftp.gnu.org/gnu/gawk/*
mawk	*ftp://ftp.whidbey.net/pub/brennan/mawk-1.3.3.tar.gz*
awka	*http://awka.sourceforge.net/*（awk转C的转换程序）

9.1 awk命令行

awk的调用可以定义变量、提供程序并且指定输入文件：

```
awk [ -F fs ] [ -v var=value ... ] 'program' [ -- ] \
    [ var=value ... ] [ file(s) ]

awk [ -F fs ] [ -v var=value ... ] -f programfile [ -- ] \
    [ var=value ... ] [ file(s) ]
```

短程序通常是直接在命令行上提供，而比较长的程序，则委托-f选项指定。遇到需连接被指名的程序文件以得到完整的程序时，则可重复使用此选项。这是包含共享awk代码的程序库之方便用法，但另一种包含程序库的方式是使用igawk程序，它是gawk分发的一部分。选项需置于文件名以及一般*var=value*赋值的前面。

如果命令行未指定文件名，则awk会读取标准输入。

--是特殊选项：指出awk本身已没有更进一步的命令行选项。任何接下来的选项都可被你的程序使用。

-F选项是用来重新定义默认字段分隔字符，且一般惯例将它作为第一个命令行选项。紧接在-F选项后的fs参数是一个正则表达式，或是被提供作为下一个参数。字段分隔字符也可设置使用内建变量FS所指定的（见本章稍后的表9-3）：

```
awk -F '\t' '{ ... }' files FS="[\f\v]" files
```

以上面的例子来看：-F选项设置的值，应用到第一个文件组，而由FS指定的值，则应用到第二个组。

初始化的 -v 选项必须放在命令行上直接给定的任何程序之前，它们会在程序启动之前以及处理任何文件之前生效。在一命令行程序之后的-v选项会被解释为一个文件名(可能是不存在的)。

在命令行上其他地方的初始化会在处理参数时完成，并且会带上文件名，例如：

```
awk '{...}' Pass=1 *.tex Pass=2 *.tex
```

处理文件的列表两次，第一次是 Pass 设为 1，第二次将它设为 2。

使用字符串值进行初始化无须用引号框起来，除非 Shell 要求这样的引用，以保护特殊字符或空白。

特殊文件名 - （连字号）表示标准输入。大部分现代的 awk 实现（但不包括 POSIX）都认定特殊名称 /dev/stdin 为标准输入，即使主机操作系统不支持该文件名。同样：/dev/stderr与/dev/stdout可用于awk程序内，分别表示标准错误输出与标准输出。

9.2 awk 程序模型

awk 把输入流看作一连串记录的集合，每条记录都可进一步细分为字段。通常，一行一条记录，而字段则由一个或多个非空白字符的单词组成。然而，是什么构成一条记录和一个字段，完全是由程序员控制，且它们的定义，甚至可以在处理期间更改。

一个 awk 程序是一对以模式（pattern）与大括号框起来的操作（action）组合而成的，或许，还会加上实现操作细节的函数（function）。针对每个匹配于输入数据的模式，操作会被执行，且所有模式都会针对每条输入记录而检查。

模式或操作可省略其中一个。如果模式省略，则操作将被应用到每条输入记录；如果操作省略，则默认操作为打印匹配之记录在标准输出上。以下是传统 awk 程序的配置：

```
pattern  { action }          如模式匹配，则执行操作
pattern                      如模式匹配，则打印记录
         { action }          针对每条记录，执行操作
```

输入会自动地由一个输入文件切换到下一个，且awk本身通常会处理每个输入文件的打开、读取与关闭，以允许用户程序专心致力于记录的处理。程序细节将在稍后的 9.5 节中详述。

虽然，模式多半是数字或字符串表达式，不过 awk 以保留字 BEGIN 与 END 提供两种特殊模式。

与BEGIN关联的操作只会执行一次，在任何命令行文件或一般命令行赋值被处理之前，

但是在任何开头的 -v 选项指定已经完成之后。它大部分是用来处理程序所需要的任何特殊初始化工作。

END操作也是只执行一次，用于所有输入数据已被处理完之后。它多半用于产生摘要报告，或是执行清除操作。

BEGIN 与 END 模式可以是任意顺序，可以存在于 awk 程序内的任何位置。不过，为了方便，我们通常将 BEGIN 模式放在程序的第一个位置，而将 END 模式放在最后。

当指定多个 BEGIN 或 END 模式，则它们将按照在 awk 程序里的顺序，一次执行。这允许使用额外的 -f 选项纳入库代码，以提供起始与清除的操作。

9.3 程序元素

就像绝大多数的脚本语言一样，awk 处理数字与字符串数据。awk 提供了标量（scalar）与数组（array）两种变量以保存数据、数字与字符串表达式，还提供了一些语句类型以处理数据：赋值、注释、条件、函数、输入、循环及输出。awk 表达式与语句的许多功能，都与 C 程序语言里的相似。

9.3.1 注释与空白

awk 里的注释是从 # 开始到该行结束，就像在 Shell 里那样。空行等同于空的注释。

语言里的任何地方都能有空白，也允许使用任何长度的空白字符，所以适时地使用空行与缩进，可以增进程序的可读性。不过，单条语句通常不能被分割跨越多行，除非在行切断的地方立即前置一个反斜杠。

9.3.2 字符串与字符串表达式

awk 里的字符串常数是以引号定界，例如："This is a string constant"。字符字符串可包含任何 8-bit 的字符，除了控制字符 NUL（字符值为 0）以外。因为 NUL 在底层实现语言（C）里，扮演的是一个字符串中断字符的角色。GNU 的 gawk 则无此限制，所以 gawk 可以安全地处理任意二进制文件。

awk 字符串包含零至多个字符，且在字符串的长度上没有限制，视可用内存而定。字符串表达式赋值给变量后，会自动建立一个字符串，且变量的前一个字符串值所占用的内存也会自动回收。

反斜杠转义序列允许非打印字符的表示，如 2.5.3 节介绍的 echo 命令一样。"A\tZ" 包

含的是：字符A、制表字符（tab），以及字符Z；而"\001"与"\x01"则每个只是包含了Ctrl-A字符。

echo不支持十六进制的转义序列，但1989年的ISO C标准里，已将此功能纳入awk实现中。不同于至多只用三个数字表示的八进制转义序列的是：十六进制转义会耗用所有接下来的十六进制数字。gawk与nawk遵循C标准，但mawk不是：它收集至多两个十六进制数字，将"\x404142"减少为"@4142"，而非成为8-bit值0x42 = 66，其表示ASCII字符集里"B"的位置。POSIX awk完全不支持十六进制转义序列。

awk提供了许多方便好用的内建函数，可在字符串上执行；我们将在9.9节中讨论。现在，我们只简略介绍字符串长度函数：length(*string*)返回*string*内的字符数。

字符串的比较，用的是传统的关系运算符：==（相等）、!=（不等）、<（小于）、<=（小于等于）、>（大于），以及>=（大于等于）。比较后返回1为真，0为假。比较不同长度的字符串，且其中一个字符串为另一个的初始子字符串时，较短的会定义为小于较长的那个，因此，"A" < "AA"值为真。

不同于大多数的程序语言拥有字符串数据类型：awk并无特殊的字符串接续运算符。也就是说，两个连续字符串，会自动地连接在一起。以下每一组赋值设置标量变量s为相同的具有四个字符的字符串：

```
s = "ABCD"
s = "AB" "CD"
s = "A" "BC" "D"
s = "A" "B" "C" "D"
```

字符串不需要是常数，如果我们继续上述的赋值：

```
t = s s s
```

则t的值为"ABCDABCDABCD"。

将数字转换为字符串，通过数字连接空字符串即可：n = 123，接着是s = "" n，把值"123"赋给s。当数字无法确切地表示时，会出现一些警告，我们将在稍后的9.9.8节说明数字转换为字符串的细节。

awk功能强大的地方大多来自于它对正则表达式的支持。有两个运算符：~（匹配）与!~（不匹配）让awk更容易使用正则表达式："ABC" ~ "^[A-Z]+$"结果为真，因为左边的字符串里只有大写字符，而右边表达式匹配任何的（ASCII）大写字母字符串。你可以到3.2.3节了解awk对扩展正则表达式（Extended Regular Expressions，ERE）的支持。

正则表达式常量可以用引号或斜杠加以定界："ABC" ~ /^[A-Z]+$/ 等同于上述的例子。要使用哪一种，根据程序员的喜好而定，不过斜杠形式是较常见的，因为它可以用来强调括起来的就是正则表达式，而非任意的字符串。然而，在极少的情况下，使用斜杠定界字符会与除号运算符相混淆，这时使用引号较好。

如果在引号字符串里正好需要有字面意义的引号，则应以反斜杠("...\"...")保护，同理，字面上的斜杠如果出现在以斜杠定界的正则表达式里，也应这么做(/...\/.../)。如果需要在正则表达式里使用反斜杠时，则它也应被保护，但引号形式则需要额外层级的保护："\\\\TeX" 与 /\\TeX/ 都匹配于包含 \TeX 的字符串的正则表达式。

9.3.3 数字与数值表达式

所有awk里的数字，都以双精确度的浮点值表示，我们已附上部分详细数据。虽然你或许并不想成为浮点算术专家，不过了解计算机算术的限制也是很重要的，因为这么一来，你就不会期待计算机无法做到的运算，也可以避免掉入一些陷阱。

浮点数可以包含一个末端以字母 e（或 E）所表示的10次方指数以及可选地带正负号的一个整数。举例来说：0.03125、3.125e-2、3125e-5 与 0.003125E1，同样都是表示1/32。因为 awk 里所有算术都是浮点算术，所以表达式 1/32 写成这种方式，就不需要担心像使用整数数据类型的程序语言中所碰到的那样计算为零的情况。

awk 并没有提供字符串转数字的函数，不过 awk 的做法很简单：只要加个零到字符串里，例如：s = "123"，接着是 n = 0 + s，便将数字 123 赋值给 n 了。

通过把这样的字符串转换为数字："+123ABC" 转换为 123，而 "ABC"、"ABC123"，与 "" 则全转换成 0，就可以强制非数值字符串转换为数字。

浮点数的有限精度，意指有些值无法准确地表示：计算的次序很重要（浮点算术没有结合性），且计算的结果通常也只是尽可能地表示为最接近的数字。

浮点数的有限范围，意指太小或太大的数字都无法表示。在现代系统中，这样的值会被转换为零和无限大。

即使awk里所有的数值运算都在浮点算术内完成，整数值还是可以确实地表示，只要值不是太大。在 IEEE 754 算术中，53 位有意义的位数，将整数至多限制在 2^{53} = 9 007 199 254 740 992。这个数值，对大部分涉及计数功能的文本处理应用程序来说已经够用了。

awk 内的数值运算符和许多其他程序语言里的相似，我们将它们总结在表 9-2 中。

浮点算术更多内容

事实上，现今所有平台已一致遵循 1985 年的《IEEE 754 Standard for Binary Floating-Point Arithmetic》。该标准定义了 32 位单精度格式、64 位双精度格式，以及可选用的扩展精度格式，通常可实现在 80 或 128 位内。awk 实现使用了 64 位格式（对应于 C 的 double 数据类型），尽管 awk 致力于可移植性地使用，但 awk 语言规格仍故意地对细节部分保持模糊。POSIX 的 awk 声称，只有算术会遵循 ISO C Standard，它不需要任何特定的浮点架构。

IEEE 754 64 位双精度的值拥有一个正负号位、一个 11 位偏移指数，以及一个 53 位的显着位（不存储开头位）。这可表示至多 16 位的十进制数字。最大的有限大小约为 10^{+308}，及最小的正规化非零值大小约为 10^{-308}。大部分 IEEE 754 实现也支持正常值以下的数值，可向下扩展范围到 10^{-324}，但会漏失精度：这种逐渐下溢（gradual underflow）到零的现象，拥有一些好用的数值型特质，不过与非数值型软件无关。

由于正负号位是显式地指定，所以 IEEE 754 算术对正零与负零都支持。许多程序语言在这方面都有所误解，awk 也不例外：部分 awk 实现在打印负零时不会带上负号。

IEEE 754 算术也包括两个特殊值：Infinity（无限大）与 NaN（not-a-number；非数字）。这两种都可以赋予正负号，不过 NaN 的正负号没有意义。它们主要是要让高性能计算机里的运算不要中断，而仍能继续记录异常情况（exceptional condition）的发生。当值太大而无法表示时，它会说溢出（overflow），并让结果为 Infinity。当值未正确地定义，例如 Infinity - Infinity，或 0/0，则结果为一 NaN。

Infinity 与 NaN 的计算：Infinity + Infinity 与 Infinity * Infinity 都产生 Infinity；而 NaN 结合任何值，都产生 NaN。

相同正负号的 Infinity 在比较时是相等的；NaN 比较时则不等于它自己：如果 x 为 NaN，则 (x != x) 的测试值为真。

awk 在 IEEE 754 算术变成广泛可用之前即已开发，因此该语言无法完整地支持 Infinity 与 NaN。特别需要说明的是：现行 awk 实现会捕捉试图除以零的陷阱，即便该运算在 IEEE 754 演算里的定义是极为完备的。

表 9-2：awk 的数值运算符（优先级由大到小排列）

运算符	说明
++ --	增加与减少（前置或后置）
^ **	指数（右结合性）
! + -	非、一元（unary）加号、一元减号
* / %	乘、除、余数
+ -	加、减
< <= == <= != > >=	比较
&&	逻辑 AND（简写）
||	逻辑 OR（简写）
? :	三元条件式
= += -= *= /= %= ^= **=	赋值（右结合性）

awk和大部分程序语言一样，可使用括号以控制计算顺序。能记得运算符确切顺序的人其实不多，特别是那些使用好几种语言的人：当有存疑时，就用括号吧！

增加与减少运算符的工作（如Shell里的一样）详见6.1.3节。分开来看：n++ 与 ++n 是一样的，但由于它们在更新变量值及返回值时，会有副作用（side effect），所以当它们在同一个语句里使用一次以上时，可能会出现难以判断的计算顺序，例如：表达式 n++ + ++n 由实现期定义。虽然有这种模棱两可的问题，但是增加/减少运算符仍广泛地使用在提供这一功能的程序语言中。

取幂运算是由左边的运算数（operand）乘幂右运算数的次方。因此，n^3 与 n**3 都指 n 的立方。这两个运算符名称是同义的，只是它们是来自不同的前身语言。惯用C的程序员可能发现：awk 的 ^ 运算符不同于在C里的，不过 awk 中大部分运算符的使用仍类似于C。

取幂与赋值 awk 里仅有的右结合性的运算符，因此：a^b^c^d 意即 a^(b^(c^d))；然而 a/b/c/d 表示的是：((a/b)/c)/d。这些结合性规则，在许多程序语言中都很常见，同时也是数学的惯例。

在最初的 awk 规范下，余数运算里如果有任一运算数为负，则余数运算符的结果是由实现期定义。POSIX awk 要求其行为要像 ISO Standard C 函数 fmod() 那样。也就是当 x % y 可表示时，则会要求这个表达式 x 带有正负号，且必须小于 y。我们测试过的所有 awk 实现，都遵循这一 POSIX 的要求。

逻辑运算符 && 与 || 与在 Shell 里的一样，为 AND 与 OR 的简写：它们只在需要的时候才计算它们右边的运算数。

表9-2中倒数第二行为简写的三元条件运算符。如果第一个运算数非零（为真），则结果为第二个运算数；否则，则为第三个运算数。第二与第三个运算数只有一个被计算。因此，在awk里，你可以写一个简洁的赋值：`a = (u > w) ? x^3 : y^7`，这种写法，在其他程序语言中，可能如下所示：

```
if (u > w) then
    a = x^3
else
    a = y^7
endif
```

赋值运算符可能会不正常，原因有二。第一：复合式，像`/=`这样，以左边运算数作为右边的第一个运算数：`n /= 3`便是`n = n / 3`的简写。第二：赋值的结果用来作为另一个表达式的部分表达式：`a = b = c = 123`先赋值123给c（因为赋值运算符为右结合性），然后再将c的值给b，最后把b的值给a。结果如预期：a、b与c都接收到值123。相同地，`x = (y = 123) + (z = 321)`分别将x、y与z指定为444、123与321。

`**`与`**=`运算符非POSIX awk的一部分，mawk也不认可。你应该避免在新的程序里再使用它，请以`^`与`^=`取代之。

注意： 请确定你了解赋值用的`=`与相等测试用的`==`是不同的。因为赋值是有效的表达式，所以`(r = s) ? t : u`表达式在句子结构上是正确的，但可能会出现不是你想要的结果。它指赋值s给r，然后如果其值非零，则返回t，否则，则返回u。这里的警告，也适用在C、C++、Java及其他同时支持`=`与`==`运算符的语言。

内建函数`int()`返回其参数的整数值部分，例：`int(-3.14159)`计算值为−3。

awk提供了一些通用的基本数学函数，可能和你用过的计算程序或其他程序语言里使用的类似，例如`sqrt()`、`sin()`、`cos()`、`log()`、`exp()`等等。详见9.10节。

9.3.4 标量变量

保存单一值的变量叫做标量变量。awk就像绝大多数的脚本语言一样：变量无须先行声明。相反地，它们会在程序里第一次使用它的时候，自动被建立，这通常是通过指定其值达成，这个值可以是数字或是字符串。当使用变量时，在内容中期待的是数字还是字符串就很清楚了，且其值也会在需要时自动地由其中一种（数字或字符串）转换为另一种。

所有的awk变量在建立时其初始值为一个空字符串值，但是当需要数值时，它会被视为零。

awk的变量名称必须以ACSII字母或下划线开始，然后选择性地接上字母、下划线及数字。因此，变量名称要匹配正则表达式**[A-Za-z_][A-Za-z_0-9]***。除此之外，变量名称在实际上并没有长度的限制。

另外，awk的变量名称是与大小写有关的：foo、Foo与FOO是完全不同的三个名称。一般使用上以及建议用法是：养成习惯，将局部变量全设为小写、全局变量第一个字母为大写，而内建变量则全是大写。

awk提供许多内建变量，都是大写名称。我们在简易程序里时常需要用到的几个，列于表9-3中。

表9-3：awk里一般常用到的内建标量变量

变量	说明
FILENAME	当前输入文件的名称
FNR	当前输入文件的记录数
FS	字段分隔字符（正则表达式）（默认为：" "）
NF	当前记录的字段数
NR	在工作（job）中的记录数
OFS	输出字段分隔字符（默认为：" "）
ORS	输出记录分隔字符（默认为："\n"）
RS	输入记录分隔字符（仅用于gawk与mawk里的正则表达式）（默认为："\n"）

9.3.5 数组变量

awk里的数组变量遵循与标量变量里相同的命名惯例，只不过它包含零到多个数据项，通过紧接着名称的数组索引选定。

大部分程序语言都需要以整数表达式作为索引的数组，但awk允许在数组名称之后，以方括号将任意数字或字符串表达式括起来作为索引。如果你先前未看过这样的数组，可能会觉得难以理解，下面的awk代码以办公室名录程序来解释，让你更容易了解其功用：

```
telephone["Alice"] = "555-0134"
telephone["Bob"]   = "555-0135"
telephone["Carol"] = "555-0136"
telephone["Don"]   = "555-0141"
```

以任意值为索引的数组，称之为关联数组，因为它们的名称与值是相关联的，就像人类所做的一样。重要的是，awk将其应用于数组中，允许查找（find）、插入（insert）以及删除（remove）等操作，在一定的时间内完成，与存储多少项目无关。

awk 里的数组无须声明也无须配置：数组的存储空间在引用新元素时会自动增长。数组存储空间是稀疏的（sparse）：只有那些确实被引用到的元素才会配置。即你可以在 x[1] = 3.14159 后面接 x[10000000] = "ten million"，而不必填满元素 2 到 9999999。绝大多数的程序语言在使用数组时，要求所有元素是相同类型，不过 awk 数组并无此限制。

当元素不再需要时，其存储空间可回收再利用。delete *array[index]* 会从数组中删除元素，而近期的 awk 实现则允许以 delete *array* 删除所有的元素。我们将在稍后 9.9.6 节中说明另一种删除数组元素的方式。

一个变量不能同时用作标量变量和数组变量。当你应用 delete 语句删除数组的元素（element）的时候，不会删除它的名称。因此，像这样的代码：

```
x[1] = 123
delete x
x = 789
```

会引发 awk 发出提示，告诉你不可以给数组名称赋值。

有时，需要使用多个索引找出表格数据中的唯一值，例如邮局使用的门牌号码、街名及邮政编码，以识别邮件递送位置。成对的列/栏可以识别二维网格内的位置，例如象棋盘。参考书目通常会记录作者、书名、编辑、出版社和出版年份，以识别一本特定书籍。鞋店店员需要知道制造商、型号、颜色及大小，才能从仓库中为顾客找到他想要的鞋子。

awk 通过将“以逗点分隔的索引列表”看作一个字符串，而使用多个索引模拟数组。然而，由于逗点本身也可能出现在索引值内，因此 awk 使用存储在内建变量 SUBSEP 里的无法打印字符串取代索引分隔字符（逗点）。POSIX 宣称它的值是由实现期定义，一般来说，其默认值为 "\034"（ASCII 字段分隔控制字符，FS），但你如果需要在索引值里使用该字符串，则你可以更改它。因此，当你写 maildrop[53, "Oak Lane", "T4Q 7XV"]，awk 会将索引列表转换为字符串表达式 "53" SUBSEP "Oak Lane" SUBSEP "T4Q 7XV"，且使用它的字符串值作为索引。这样的结构是可以被推翻的，不过我们不建议你这样做，下面这些语句都显示相同的项目：

```
print maildrop[53, "Oak Lane", "T4Q 7XV"]
print maildrop["53" SUBSEP "Oak Lane" SUBSEP "T4Q 7XV"]
print maildrop["53\034Oak Lane", "T4Q 7XV"]
print maildrop["53\034Oak Lane\034T4Q 7XV"]
```

很明显，如果你稍后改变了 SUBSEP 的值，将会使得已经存储数据的索引失效，所以，SUBSEP 其实应该在每个程序里只设置一次，在 BEGIN 操作里。

一旦适当地重新调整思路，利用关联数组，可以解决许多数据处理的问题。针对像 awk 这样的简单程序语言来说，它们已经展现了自己是一个优秀的设计选择。

9.3.6 命令行参数

awk对于命令行的自动化处理，意味着awk程序几乎不需要关心它们自己。这点与C、C++、Java以及Shell的处理全然不同，使用后面这些程序的设计师，习惯于明白确切地处理命令行参数。

awk通过内建变量ARGC（参数计数）与ARGV（参数向量，或参数值），让命令行参数可用。下面简短的程序说明其用法：

```
$ cat showargs.awk
BEGIN {
    print "ARGC =", ARGC
    for (k = 0; k < ARGC; k++)
        print "ARGV[" k "] = [" ARGV[k] "]"
}
```

再来看看将它用在一般awk命令行上，会产生什么样的结果：

```
$ awk -v One=1 -v Two=2 -f showargs.awk Three=3 file1 Four=4 file2 file3
ARGC = 6
ARGV[0] = [awk]
ARGV[1] = [Three=3]
ARGV[2] = [file1]
ARGV[3] = [Four=4]
ARGV[4] = [file2]
ARGV[5] = [file3]
```

正如在C与C++中：参数存储在数组项目0、1、…、ARGC－1中，且第0个项目是awk程序本身的名称。不过，与-f与-v选项结合性的参数是不可使用的。同样的，任何命令行程序也不可使用：

```
$ awk 'BEGIN { for (k = 0; k < ARGC; k++)
>     print "ARGV[" k "] = [" ARGV[k] "]" }' a b c
ARGV[0] = [awk]
ARGV[1] = [a]
ARGV[2] = [b]
ARGV[3] = [c]
```

是否需要显示在程序名称里的目录路径，则看实际情况而定：

```
$ /usr/local/bin/gawk 'BEGIN { print ARGV[0] }'
gawk

$ /usr/local/bin/mawk 'BEGIN { print ARGV[0] }'
mawk

$ /usr/local/bin/nawk 'BEGIN { print ARGV[0] }'
/usr/local/bin/nawk
```

awk程序可修改ARGC与ARGV，尽管极少需要这么做。如果ARGV的元素被（重新）设置为空字符串或被删除，则awk会忽略它，不会将它视为文件名。如果你消除ARGV的结尾几个项目，请确定相应地减少ARGC。

awk一见到参数含有程序内容或是特殊--选项时，它会立即停止将参数解释为选项。任何接下来看起来像是选项的参数，都必须由你的程序处理，并接着从ARGV中被删除，或设置为空字符串。

时常我们会在Shell脚本中包裹awk引用。为保持脚本的可读性，可将一段冗长的程序存储在Shell变量里。你也可以将脚本一般化：通过具有nawk默认值的环境变量，以允许在执行时可选择awk实现：

```
#! /bin/sh -
AWK=${AWK:-nawk}
AWKPROG='
    ... 这里是长的程序 ...
'
$AWK "$AWKPROG" "$@"
```

单引号可保护程序内容不被Shell解释，不过当程序本身包含单引号时，你得更小心处理。另一种在Shell变量里存储程序的好用替代方式是：将它放进共享程序库目录内个别的文件内，这个程序库目录可在相对于存储此脚本的目录中找到：

```
#! /bin/sh -
AWK=${AWK:-nawk}
$AWK -f `dirname $0`/../share/lib/myprog.awk -- "$@"
```

dirname命令已在8.2节里作过说明。举例来说，如果脚本在/usr/local/bin下，那么程序就在/usr/local/share/lib下。这里的dirname是用来确保只要两个文件的相对位置被保留，则脚本将可运行。

9.3.7 环境变量

awk提供访问内建数组ENVIRON中所有的环境变量：

```
$ awk 'BEGIN { print ENVIRON["HOME"]; print ENVIRON["USER"] }'
/home/jones
jones
```

ENVIRON数组并无特别之处：你可以依需要来加入、删除及修改项目。然而，POSIX要求子进程继承awk启动时生效的环境，而我们也发现，在现行实现下，并无法将对于ENVIRON数组的变更传送给子进程，或者是给内建函数。特别地，这是指你无法通过对于ENVIRON["LC_ALL"]的更改控制字符串函数，例如tolower()，在特定locale下的行为模式。因此，你应将ENVIRON看成是一个只读数组。

如果你需要控制子进程的locale，则可通过在命令行字符串里设置适合的环境变量达成。例如，以 Spanish（西班牙语）的 locale 排序文件，像这样：

```
system("env LC_ALL=es_ES sort infile > outfile")
```

system()函数将在稍后 9.7.8 节中说明。

9.4 记录与字段

在awk程序化模式中，通过输入文件隐含循环的每一次迭代，会处理单一记录(record)，通常是一行文本。记录可进一步再分割为更小的字符串，叫做字段（field)。

9.4.1 记录分隔字符

尽管记录通常是被换行字符所分隔的数行文本，但awk允许更具通用性地通过记录分隔字符内建变量 RS。

在传统的与POSIX的 awk 里，RS必须是单一字面字符，例如换行符号（默认值），或是空字符串。后者会被特殊处理：记录是由一个或多个空行所分隔的段落，且位于文件起始或结尾处的空行会被忽略。字段则再由换行字符，或 FS 里所设置的任何字符加以分隔。

gawk与mawk提供了一个重要的扩展功能：RS可以是正则表达式，也就是提供比单一字符还长的长度。因此，RS = "+" 匹配于字面上的一个加号，然而 RS = ":+" 匹配于一个或多个冒号。这提供了更强大的记录规格，我们在9.6节中，将此应用到部分范例。

使用正则表达式记录分隔字符时，匹配分隔字符的文本不再是由RS值来决定。gawk以内建变量 RT 中的语言扩展来提供此功能，mawk 则不支持。

如果没有 RS 的正则表达式扩展，当它们遇到要匹配跨行的情况时，要将正则表达式模拟成记录分隔字符是很难的，因为绝大多数的 UNIX 文本处理工具是一次处理一行。有时，你可以使用 tr 将换行符号转换为其他未用到的字符，让流串成极长的一行。然而，这时常会遇到其他工具程序里缓冲区大小限制的冲突。gawk、mawk 和 emacs 是少数几个没有面向行的数据浏览限制的程序。

9.4.2 字段分隔字符

字段彼此是被匹配字段分隔字符正则表达式（可在变量 FS 里取得）的当前字符串值分隔。

FS的默认值为单一空格(space),它接受特殊的解释方式:一个或多个空白字符(空格与制表字符)以及行的开头与结尾的空白,都将被忽略。因此,当输入行为:

```
alpha beta gamma
    alpha     beta      gamma
```

这两行对于使用FS默认值的awk程序来说是一样的:具有值"alpha"、"beta"与"gamma"的三个字段。当数据是由人为输入时,这个功能特别方便。

当使用单一空格分隔字段的少有情况时,只要设置FS = "[ ]"以正好匹配一个空格即可。在这样的设置下,前置与结尾的空白不会再被忽略。而下面这两个例子也会报告不同的字段数(输入记录的开头与最结尾各有两个空格):

```
$ echo '  un deux trois  ' | awk -F' ' '{ print NF ":" $0 }'
3:  un deux trois

$ echo '  un deux trois  ' | awk -F'[ ]' '{ print NF ":" $0 }'
7:  un deux trois
```

第二个例子看到了七个字段:""、""、"un"、"deux"、"trois"、""以及""。

FS只有在它超过一个字符时,才会被视为正则表达式。FS = "."指的是以.作为字段分隔字符;而不是正则表达式所指的任何单一字符。

现代的awk实现也允许FS为一个空字符串,每一字符均为一个分开的字段。但是旧式的实现,则解读成每条记录只有一个字段。POSIX宣称,这种行为仅在未标明空字段分隔字符时有效。

9.4.3 字段

字段可以特殊名称$1、$2、$3、…、$NF供awk程序使用。字段引用无须是固定的,有必要的话,它们还可以转换(通过截断)为整数值:假定k为3,则值$k、$(1+2)、$(27/9)、$3.14159、$"3.14159",以及$3,都引用到第三个字段。

特殊字段名称$0引用到当前记录,初始值是从输入流中读取,且记录分隔字符不是记录的一部分。引用到0到NF范围以上的字段编号是不会有错:它们会返回空字符串,且不会建立新字段,除非你指定值给它们。如引用到分数或非数字,则字段编号是在实现期定义的。引用负值字段编号则会发生极严重的错误,在我们测试过的所有实现下都是这样。POSIX宣称引用到任何非负数以外的字段编号都是未指定的。

字段就如同一般变量,也可赋值,例如$1 = "alef"是合法的,但会有一个较大的副

作用：如果连续引用完整的记录，则它从字段的当前值中重新组合在一起，但是由输出字段分隔符的内建变量 OFS 给定字符串的分隔，默认为单一空格。

9.5 模式与操作

模式与操作构成awk程序的核心。awk的非传统数据驱动程序模式，使得它更吸引用户使用，也成就了许多 awk 程序的简洁形式。

9.5.1 模式

模式由字符串与/或数值表达式构建而成：一旦它们计算出当前输入记录的值为非零（真），则实行结合性的操作。如果模式是正则表达式，则意指此表达式会被拿来与整个输入记录进行匹配，就好像你已经写 $0 ~ /regexp/ 而非只是 /regexp/。下面例子是选定模式时，常用到的几种形式：

```
NF == 0                                   选定空记录
NF > 3                                    选定拥有三个字段以上的记录
NR < 5                                    选定第 1 到第 4 条记录
(FNR == 3) && (FILENAME ~ /[.][ch]$/)     于 C 来源文件中选定记录 3
$1 ~ /jones/                              选定字段 1 里有 "jones" 的记录
/[Xx][Mm][Ll]/                            选定含有 "XML" 的记录，并忽略大小写差异
$0 ~ /[Xx][Mm][Ll]/                       同上
```

awk 在匹配功能上，还可以使用范围表达式（range expression）。以逗点隔开的两个表达式，会从匹配于左边表达式处（含）开始取样，直到匹配右边的表达式。如果两个范围表达式匹配后匹配于一条记录，则选定该单一记录。这种行为不同于 sed，sed 仅在连续起始范围记录之后的那些记录里，查找范围的结尾模式。这里有几个例子：

```
(FNR == 3), (FNR == 10)                           选定每个输入文件里的记录 3 到 10
/<[Hh][Tt][Mm][Ll]>/, /<\/[Hh][Tt][Mm][Ll]>/      选定 HTML 文件里的主体
/[aeiouy][aeiouy]/, /[^aeiouy][^aeiouy]/          选定起始于两个元音、结尾为两个
                                                  辅音的记录
```

在 BEGIN 操作里，FILENAME、FNR、NF 与 NR 初始都未定义；引用到它们时，会返回 null 字符串或零。

如果程序里仅包括 BEGIN 模式的操作，则 awk 会在完成最后一个操作之后退出，而不需读取任何文件。

进入第一个 END 操作时，FILENAME 是最后一个要处理的输入文件，而 FNR、NF 和 NR 则会保留它们从最后一条输入记录而来的值。在 END 操作里的 $0 值是不可靠的：gawk 与 mawk 会保留它，但 nawk 不会，而 POSIX 则是静默。

9.5.2 操作

到目前为止，我们已提及了在选定记录时要用到的大部分awk语言元素。而操作段落则是可选地接在一个模式之后，也就是操作所在之处：它标明了如何处理该记录。

awk提供许多语句类型，可允许使用任意程序的构建。不过，我们要到后面的9.7节才会详细说明这部分，在这里，除了指定语句之外，我们只考虑简单的print语句。

以最简单的形式来说，纯print意指在标准输出上，打印当前的输入记录（$0），接着是输出记录分隔字符ORS的值，默认为单一换行字符。因此，下面这些程序所做的全是相同的操作：

```
1                                     模式为真，默认操作为打印
NR > 0  { print }                     有记录时打印（恒为真）
1       { print }                     模式为真，则打印，这是默认值
        { print }                     无模式则视为真，明确的打印，这是默认值
        { print $0 }                  相同，但打印明确的值
```

含有上述任何一行的单行awk程序，只会将输入流复制到标准输出。

更常见的用法是：一个print语句里包含了以逗点隔开的零或多个表达式。每个表达式会被计算，有必要时会转换为一个字符串，且以输出字段分隔字符OFS的值将输出分隔后传送到标准输出。接在最后项目之后的是输出记录分隔字符ORS的值。

print的参数列表及类似功能的printf与sprintf，参见9.9.8节，都可选地放置到圆括号内。圆括号的使用，是为了避免参数列表包含关系运算符时被误解，因为像<与>也可用于I/O重定向，详见9.7.6节与9.7.7节。

下面的例子已经是完整的awk程序。在每一个中，我们都只显示前三个输入字段，并通过省略选定模式，选定所有的记录。awk程序语句以分号分隔，而且我们会使用些略微不同的操作代码，以修改输出字段分隔字符：

```
$ echo 'one two three four' | awk '{ print $1, $2, $3 }'
one two three

$ echo 'one two three four' | awk '{ OFS = "..."; print $1, $2, $3 }'
one...two...three

$ echo 'one two three four' | awk '{ OFS = "\n"; print $1, $2, $3 }'
one
two
three
```

改变输出字段分隔字符而没有指定任何字段，不会改变$0：

```
$ echo 'one two three four' | awk '{ OFS = "\n"; print $0 }'
one two three four
```

不过，如果我们更改输出字段分隔字符，并指定至少一个字段（即使我们未变更其值），强制以新的字段分隔字符重新组合记录，则结果为：

```
$ echo 'one two three four' | awk '{ OFS = "\n"; $1 = $1; print $0 }'
one
two
three
four
```

9.6 在 awk 里的单行程序

我们到此已介绍了在awk中使用单行程序所完成的很多任务；很少有其他程序语言可以做到这样。在本节，我们会再介绍一些这类单行程序的例子，不过由于篇幅的限制，我们有时不得不将它切为数行。在这些例子里，我们会展示以awk或其他UNIX工具程序解决问题的多种方式：

- 我们从简单的awk实现开始 —— UNIX单词计数程序wc：

  ```
  awk '{ C += length($0) + 1; W += NF } END { print NR, W, C }'
  ```

  注意：模式/操作组并不需要以换行字符分隔，不过我们通常会为了阅读上的方便而这么做。虽然我们可以包括采用 `BEGIN { C = W = 0 }`形式的初始化块，不过，awk具有默认的初始化保证，因而这部分是不怎么需要的。上述程序中，C的字符计数在每条记录处被更新，计算记录的长度，并加上换行字符（默认的记录分隔字符）。在W内的单词计数会累积字段的数目。我们不需要保留一个行计数变量，因为内建记录计数NR会自动为我们追踪该信息。END操作则处理wc所产生的单行报告打印。

- 如果程序为空，则awk不会读取任何的输入并立即退出，所以我们可以匹配cat（作为一个有效率的数据槽）：

  ```
  $ time cat *.xml > /dev/null
  0.035u 0.121s 0:00.21 71.4%     0+0k 0+0io 99pf+0w
  $ time awk '' *.xml
  0.136u 0.051s 0:00.21 85.7%     0+0k 0+0io 140pf+0w
  ```

- 撇开NUL字符问题，awk其实可以轻松取代cat，下面这两个例子会产生相同输出：

  ```
  cat *.xml
  awk 1 *.xml
  ```

- 要将原始数据值及它们的对数打印为单栏的数据文件，可使用：

  ```
  awk '{ print $1, log($1) }' file(s)
  ```

- 要从文本文件里，打印5%行左右的随机样本，可使用虚拟随机产生函数（见9.10节），这会产生平均分布于0与1之间的值：

```
awk 'rand() < 0.05' file(s)
```

- 在以空白分隔字段的表格中，报告第 *n* 栏的和：

```
awk -v COLUMN=n '{ sum += $COLUMN } END { print sum }' file(s)
```

- 微调上述报告，产生字段 *n* 的平均值：

```
awk -v COLUMN=n '{ sum += $COLUMN } END { print sum / NR }' file(s)
```

- 针对花费文件（其记录包含描述与金额于最后一个字段），打印花费总数。可使用内建变量 NF 计算总值：

```
awk '{ sum += $NF; print $0, sum }' file(s)
```

- 这里是三种查找文件内文本的方式：

```
egrep 'pattern|pattern' file(s)
awk '/pattern|pattern/' file(s)
awk '/pattern|pattern/ { print FILENAME ":" FNR ":" $0 }' file(s)
```

- 如果你要限制仅查找 100 – 150 行，可以通过两个工具程序，再搭配管道，不过这么做会漏掉位置信息：

```
sed -n -e 100,150p -s file(s) | egrep 'pattern'
```

使用 GNU sed 要搭配 -s 选项，才能为每个文件重新开始行编号。另外，你也可以通过 awk，使用比较花哨的模式来做：

```
awk '(100 <= FNR) && (FNR <= 150) && /pattern/ \
       { print FILENAME ":" FNR ":" $0 }' file(s)
```

- 要在一个四栏表格里，调换第二与第三栏，假设它们是以制表字符分隔，那么可以使用下面三种方式的其中一种：

```
awk -F'\t' -v OFS='\t' '{ print $1, $3, $2, $4 }' old > new
awk 'BEGIN { FS = OFS = "\t" } { print $1, $3, $2, $4 }' old > new
awk -F'\t' '{ print $1 "\t" $3 "\t" $2 "\t" $4 }' old > new
```

- 要将各栏分隔字符由制表字符（在此以 · 显示）转换成 &，可在以下两种方式择一：

```
sed -e 's/·/\&/g' file(s)
awk 'BEGIN { FS = "\t"; OFS = "&" } { $1 = $1; print }' file(s)
```

- 下面这两个管道，都为删除已排序流里的重复行：

```
sort file(s) | uniq
sort file(s) | awk 'Last != $0 { print } { Last = $0 }'
```

- 将回车字符/换行字符的行终结，一致转换为以换行字符作为行终结，可在下列方式中选择一种：

```
sed -e 's/\r$//' file(s)
sed -e 's/^M$//' file(s)
mawk 'BEGIN { RS = "\r\n" } { print }' file(s)
```

第一个 sed 需要一个现代的版本，它会认得转义序列。在第二个例子里，^M 表示

的是字面上的Ctrl-M（回车）。第三个例子，我们需要gawk或是mawk，因为nawk与POSIX awk都不支持在RS里拥有超过单一字符以上的设置方式。

- 要将单空格的文本行，转换为双空格的行，可在下列方式选择一种：

```
sed -e 's/$/\n/' file(s)
awk 'BEGIN { ORS = "\n\n" } { print }' file(s)
awk 'BEGIN { ORS = "\n\n" } 1' file(s)
awk '{ print $0 "\n" }' file(s)
awk '{ print; print "" }' file(s)
```

正如之前一样，我们需要现代的sed版本。请留意，在第一个awk例子里，我们如何以简单变更输出记录分隔字符ORS解决此问题：程序的剩余部分只要将每条记录显示出来即可。其余的两个awk解决方案则需要对每条记录作更多的处理，且通常在执行上会比第一个慢些。

- 将双空格行转换为单空格一样是很容易的：

```
gawk 'BEGIN { RS="\n *\n" } { print }' file(s)
```

在Fortran 77的程序里，寻找超过限制长度72个字符的行（注2），可使用下列方式之一：

```
egrep -n '^.{73,}' *.f
awk 'length($0) > 72 { print FILENAME ":" FNR ":" $0 }' *.f
```

我们需要与POSIX兼容的egrep，执行扩展正则表达式，以匹配73或73个以上的任何字符。

- 为了从文件中取出国际标准书号（ISBN）里具有连字号的值，我们需要一个长的，但直觉易懂的正则表达式，辅以记录分隔字符集，以匹配所有不属于ISBN的字符：

```
gawk 'BEGIN { RS = "[^-0-9Xx]" }
/[0-9][-0-9][-0-9][-0-9][-0-9][-0-9][-0-9][-0-9][-0-9][-0-9][-0-9]-[0-9Xx]/' \
    file(s)
```

在POSIX兼容的awk下，长的正则表达式可以缩短成/[0-9][-0-9]{10}-[-0-9Xx]/。经我们测试后发现，只有gawk --posix、HP/Compaq/DEC OSF/1 awk、Hewlett-Packard HP-UX awk、IBM AIX awk及Sun Solaris /usr/xpg4/bin/awk在正则表达式中支持POSIX扩展的括号区间表达式。

- 要截去HTML文本里以角括号框起的标记标签（markup tag），可以将标签视为记录的分隔字符，像这样：

```
mawk 'BEGIN { ORS = " "; RS = "<[^<>]*>" } { print }' *.html
```

注2： Fortran行长度限制在旧式的打孔卡上不会有问题，但当它出现在现今以屏幕为主要编辑媒介上时就麻烦了，它会令编辑器静默地忽略超过72个字段宽度的语句内容，而演变成相当棘手的bug。

通过将ORS设置为一空格，会使得HTML标记被转换为一空格，且所有输入行的断行都会被保留下来。

- 下面的例子是说明：如何将一组XML文件（就像本书一样）里所有的标题（title）取出，然后打显示来，一个标题一行，并以标记包围它。即便是标题横跨数行，此程序都能正常运行，除此之外还能处理标签单词与关闭角括号之间的空格，这种情况虽不常见，但是合法的：

```
$ mawk -v ORS=' ' -v RS='[ \n]' '/<title *>/, /<\/title *>/' *.xml |
>       sed -e 's@</title *> *@&\n@g'
...
<title>Enough awk to Be Dangerous</title>
<title>Freely available awk versions</title>
<title>The awk Command Line</title>
...
```

awk程序所产生的是单行输出，所以现代版本的sed过滤程序提供必需的断行。在这里可以省去sed处理，要这么做的话，则必须用到下一段要讨论的awk语句。

9.7 语句

程序语言必须支持连续性的、条件式的及重复的执行。awk大量地借用C程序语言的语句以提供这些功能。除此之外，本节还会提到awk专有的不同语句类型。

9.7.1 连续执行

连续的执行是以一个语句一行或以分号隔开的方式，提供一连串语句列表。下面这三行：

```
n = 123
s = "ABC"
t = s n
```

也可以这样写：

```
n = 123; s = "ABC"; t = s n
```

在单行程序里，我们通常会使用分号形式，但awk程序也支持由文件提供的方式，我们通常会将各个语句分别放在它自己的行上，反而很少用到分号。

虽然程序预期的是单一语句，不过我们还是能使用复合语句（compound statement）的方式，以大括号将语句组起来。因此，和awk模式相关联的操作正是复合语句。

9.7.2 条件式执行

awk以if语句提供条件式的执行：

```
if (expression)
    statement1

if (expression)
    statement1
else
    statement2
```

如果 *expression* 非零（为真），则执行 *statement1*。否则，如果有 else 则执行 *statement2*。这里的每个语句本身也可以是 if 语句，所以多分支条件语句的一般写法通常是这样：

```
if (expression1)
    statement1
else if (expression2)
    statement2
else if (expression3)
    statement3
...
else if (expressionk)
    statementk
else
    statementk+1
```

最后一个 else 是选择性的，它一定与前一个最接近的 if（在相同层级上）相关联。

在多分支的 if 语句里，是依次测试条件表达式：如果第一个就匹配，则选定相关联的语句执行，之后，控制权继续执行接在完整 if 语句后面的语句，无须计算语句剩余部分的条件式表达式。如果无表达式匹配，则执行最后的 else 分支。

9.7.3 重复执行

awk 提供了 4 种重复执行语句（循环）：

- 循环在起始处使用结束测试：

```
while (expression)
    statement
```

- 循环在结尾处使用结束测试：

```
do
    statement
while (expression)
```

- 循环执行可计数的次数：

```
for (expr1; expr2; expr3)
    statement
```

- 循环处理关联数组里的元素：

```
for (key in array)
    statement
```

while 循环可满足重复许多次的需求，典型的情况就是：当有数据时，就处理它。do 循环则较少用到：例如，它通常出现在优化问题的地方，用来减少“错误计算的计算，且当错误太大时可以不断地重复测试”。这两个循环都是在表达式为非零（真）的情况下进行。如果表达式初始值为零，则 while 的循环体完全不会执行，然而 do 循环体则执行一次。

for 循环的第一种形式包含三个以冒号分隔的表达式，其中任一个或是全部都可为空。第一个表达式于循环开始之前被计算。第二个则于每次重复的起始时被计算，且当它是非零（为真）时，循环会继续下去。第三个表达式是在每个重复结束时被计算。传统循环是从 1 到 *n*，写法就像这样：

```
for (k = 1; k <= n; k++)
    statement
```

然而，索引不一定需要一次重复就加一。循环也可以倒着执行，像这样：

```
for (k = n; k >= 1; k--)
    statement
```

注意： 因为浮点算术通常不精确，所以请避免在 for 语句表达式里，计算非整数的值。例如这类循环：

```
$ awk 'BEGIN { for (x = 0; x <= 1; x += 0.05) print x }'
...
0.85
0.9
0.95
```

在最后的重复中不会显示 1，因为增加的部分为不精确的表示值 0.05，因此最后所产生的 x 值会比 1.0 大一些。

C 程序员应该会发现，awk 缺乏逗点运算符，所以这三种 for 循环表达式都无法以逗点作为表达式列表的分隔。

for 循环的第二种形式，用来反复处理数组里的元素，可用在元素数量未知或未形成可运算的整数序列时。元素可以任意顺序被选定，所以输出如下：

```
for (name in telephone)
    print name "\t" telephone[name]
```

这不太可能出现你所想要的顺序。我们会在 9.7.7 节里介绍如何解决这类问题。split() 函数在 9.9.6 节里会介绍如何处理多索引的数组。

正如Shell里的模式：break语句可用于提早退出最内部的循环：

```
for (name in telephone)
    if (telephone[name] == "555-0136")
        break
print name, "has telephone number 555-0136"
```

不过，Shell风格的多层次break *n*语句在这里并不支持。

continue语句就像在Shell那样，会跳到循环体的结尾，准备执行下一个重复。awk不接受Shell的多层continue *n*语句。为了说明continue语句，我们在例9-1所展示的程序中，通过强制测试除数，以找出一数字是否为复数（composite）还是质数（记得吗？质数指的就是无法被1及其本身以外的任何数整除的数），然后显示它可找到的任何因数分解。

例 9-1：整数的因数分解

```
# 计算整数的因数分解，一行列出一个
# 语法：
#         awk -f factorize.awk
{
    n = int($1)
    m = n = (n >= 2) ? n : 2
    factors = ""
    for (k = 2; (m > 1) && (k^2 <= n); )
    {
        if (int(m % k) != 0)
        {
            k++
            continue
        }
        m /= k
        factors = (factors == "") ? ("" k) : (factors " * " k)
    }
    if ((1 < m) && (m < n))
        factors = factors " * " m
    print n, (factors == "") ? "is prime" : ("= " factors)
}
```

请留意，循环变量k是增加的，而continue语句只有在我们找到k不是m的除数时，才会执行，所以for语句中的第三个表达式为空。

如果我们使用适当的测试数据执行它，会得到以下结果：

```
$ awk -f factorize.awk test.dat
2147483540 = 2 * 2 * 5 * 107374177
2147483541 = 3 * 7 * 102261121
2147483542 = 2 * 3137 * 342283
2147483543 is prime
2147483544 = 2 * 2 * 2 * 3 * 79 * 1132639
2147483545 = 5 * 429496709
```

```
2147483546 = 2 * 13 * 8969 * 9209
2147483547 = 3 * 3 * 11 * 21691753
2147483548 = 2 * 2 * 7 * 76695841
2147483549 is prime
2147483550 = 2 * 3 * 5 * 5 * 19 * 23 * 181 * 181
```

9.7.4 数组成员的测试

成员测试 *key* in *array* 是一个表达式：如果 *key* 为 *array* 的一个索引元素，则计算为 1（真）。此外，也可通过否定运算符反转测试：如果 *key* 不是 *array* 的一个索引元素，则 !(*key* in *array*) 为 1 —— 此处圆括号是必需的。

对于具有多下标（subscript）的数组，在测试时，请使用圆括号，并以逗点分隔下标列表：(*i*, *j*, ... , *n*) in *array*。

成员测试不可能建立数组元素，然而引用元素时，如果元素不存在，便会建立它。因此你应该这么写：

```
if ("Sally" in telephone)
   print "Sally is in the directory"
```

而非：

```
if (telephone["Sally"] != "")
    print "Sally is in the directory"
```

因为第二种形式会在她（Sally）不存在时，将其加入到目录里，并拥有一个空电话号码。

重点是：你必须能够区分寻找索引（index）与寻找特定值（value）的差异。索引成员测试需要固定的时间，而值的查找时间是与数组里元素的个数成正比，这点我们在先前已通过 break 语句内的 for 循环解释过了。如果你需要时常用到这两种运算，那么构建反索引数组会比较实用：

```
for (name in telephone)
    name_by_telephone[telephone[name]] = name
```

接下来，你就可以使用 name_by_telephone["555-0136"] 在一定时间内找到 "Carol"。当然，这里假定所有的值是唯一的：如果这两人共享同一个电话，则 name_by_telephone 数组只会记录最后一个名称。只要稍做修改就能解决这个问题：

```
for (name in telephone)
{
    if (telephone[name] in name_by_telephone)
        name_by_telephone[telephone[name]] = \
            name_by_telephone[telephone[name]] "\t" name
```

```
        else
            name_by_telephone[telephone[name]] = name
    }
```

现在,name_by_telephone即包含了以制表字符分隔的具有相同电话号码的人名列表。

9.7.5 其他的流程控制语句

我们已讨论过 break 与 continue 语句如何解释重复语句内的控制流程。有时，你会需要改变 awk 在匹配输入记录与模式/操作列表中的模式时之控制流程。有三种情况要处理：

只针对此记录略过更进一步的模式检查

使用 next 语句。有些实现不允许在用户定义函数中使用 next，详见 9.8 节。

针对当前输入文件略过更进一步的模式检查

gawk 与近期的 nawk 都提供 nextfile 语句，它会使得当前输入文件立即关闭，且模式的匹配会从命令行上下一个文件里的记录重新开始。

在旧式的 awk 实现下，你可以轻松模拟出 nextfile 语句，不过会牺牲一些效率就是了。你可以用 SKIPFILE = FILENAME; next 取代 nextfile 语句，然后将这些新的模式/操作组合新增到程序开始处：

```
FNR == 1                    { SKIPFILE = "" }
FILENAME == SKIPFILE        { next }
```

第一对模式/操作是在每个文件的起始处，将 SKIPFILE 重设为空字符串，这么一来，在两个连续参数出现相同文件名时，程序才能正确处理。即使仍继续从当前文件中读取记录，它们仍会马上被 next 语句忽略。当到达文件结尾且下一个输入文件被打开时，第二个模式便不再匹配了，所以在它操作里的 next 语句就不会被执行。

略过整个工作的更进一步执行，并返回状态码给 Shell

使用 exit *n* 语句。

9.7.6 用户控制的输入

awk 直接处理命令行上标明的输入文件，意指绝大多数的 awk 程序都不必自己打开与处理文件。它也可以通过 awk 的 getline 语句做这件事。例如，拼字检查程序，通常就需要载入一个或多个目录之后，才能开始运行。

getline 会返回一个值，且可以以函数的方式使用，即使它其实是一个语句，而且是具有非惯例语法的语句。当输入被成功读取时，它的返回值为 +1，而返回值为 0 时，则表示在文件结尾，而 −1 则表示错误。它的用法很多，见表 9-4。

表 9-4：getline 的各种用法

语法	说明
getline	从当前输入文件中，读取下一条记录，存入 $0，并更新 NF、NR 与 FNR。
getline *var*	从当前输入文件中，读取下一条记录，存入 *var*，并更新NR与FNR。
getline < *file*	从 *file* 中读取下一条记录，存入 $0，并更新 NF。
getline *var* < *file*	从 *file* 中读取下一条记录，存入 *var*。
cmd | getline	从外部命令 *cmd* 读取下一条记录，存入 $0，并更新 NF。
cmd | getline *var*	从外部命令 *cmd* 读取下一条记录，存入 *var*。

现在来看一些 getline 的用法。先提问，再读取与检查答案：

```
print "What is the square root of 625?"
getline answer
print "Your reply, ", answer ", is", (answer == 25) ? "right." : "wrong."
```

如果我们想要确保输入是来自于控制终端，而非标准输入，则改用：

```
getline answer < "/dev/tty"
```

接下来，从字典中载入单词列表：

```
nwords = 1
while ((getline words[nwords] < "/usr/dict/words") > 0)
    nwords++
```

命令管道在 awk 里可以发挥强大的功能。管道可以在字符字符串中标明，也可以包含任意的 Shell 命令。这里是与 getline 搭配使用，如下：

```
"date" | getline now
close("date")
print "The current time is", now
```

大部分系统会限制打开文件的个数，所以当使用管道通过时，我们通过 close() 函数关闭管道文件。在旧式 awk 实现里，close 为语句，所以没有可移植的方式，可以像使用函数一样使用它，并取得可靠的返回代码。

接下来说明的是：如何在循环里使用命令管道：

```
command = "head -n 15 /etc/hosts"
while ((command | getline s) > 0)
    print s
close(command)
```

我们使用变量保存管道，以避免重复复杂字符串，并确保所有使用的命令都确实匹配。

在命令字符串里，每个字符都是有意义的，即使是不被注意的单一空格差异，也会引用到不同的命令。

9.7.7 输出重定向

print与printf语句（见9.9.8节）多半是将其输出传送到标准输出。不过，你也可以改成传送到文件：

```
print "Hello, world" > file
printf("The tenth power of %d is %d\n", 2, 2^10) > "/dev/tty"
```

为了附加到已存在的文件（或是该文件不存在时，则建立一个新的）中，可使用>>输出重定向：

```
print "Hello, world" >> file
```

你可以在多个输出语句上，将它们的输出全部重定向到相同的文件。当完成写入输出后，请使用close(*file*)关闭文件，释放它使用的资源。

请避免在没有适当插入close()的情况下，混用>与>>传到相同文件。在awk里，这些运算符告知输出文件应该如何打开使用。一旦打开后，文件便会一直保持在打开状态，直到你明确指出要关闭它或直到程序终结。相比之下，Shell的重定向是要求每个命令打开文件并关闭它。

或者，你也可以将输出传送到管道：

```
for (name in telephone)
    print name "\t" telephone[name] | "sort"
close("sort")
```

由于输入是来自管道，因此关闭输出管道的操作是在完成时立即执行。如果你需要在相同程序中读取输出时，这点格外重要。例如，你可以指示输出到临时性文件，然后在完成之后再读取它：

```
tmpfile = "/tmp/telephone.tmp"
command = "sort > " tmpfile
for (name in telephone)
    print name "\t" telephone[name] | command
close(command)
while ((getline < tmpfile) > 0)
    print
close(tmpfile)
```

在awk里的管道，使得整个UNIX工具集可以任我们支配，避免对其他程序语言中提供大量函数库的需求，也有助于让语言保持在小规模状态。例如，awk不提供排序的内建函数，因为它只要复制功能强大的sort命令的功能即可，详见4.1节。

近期的 a w k 实现，除了 POSIX 外，都提供将缓冲区数据导到输出流的函数：fflush(*file*)。留意一开始的两个 ff（指的是 *file flush*）。它会在成功时返回 0，失败时返回 -1。调用 fflush()（省略参数）与 fflush("")（参数为空的字符串）的行为模式视执行期而定，换句话说：请避免在可移植性很重要的程序里使用它。

9.7.8 执行外部程序

我们已在之前提到过，getline语句以及在awk管道里的输出重定向都可与外部程序通信。system(*command*)函数提供的是第三种方式：其返回值为命令的退出状态码。首先，它会先清除所有缓冲区输出，然后开始一个 /bin/sh 实例并将命令送给它。Shell 的标准错误输出和标准输出与 awk 程序的相同，所以除非命令的 I/O 被重定向，否则来自 awk 程序和 Shell 命令两者的输出，都会以预期的顺序出现。

这里是解决电话名录排序问题较短的程序方案，使用临时性文件与system()，而非awk 管道：

```
tmpfile = "/tmp/telephone.tmp"
for (name in telephone)
    print name "\t" telephone[name] > tmpfile
close(tmpfile)
system("sort < " tmpfile)
```

临时性文件必须在调用 system()之前关闭，以确保任何缓冲区输出都正确地记录在文件内。

对于被 system() 执行的命令并不需要调用 close()，因为 close() 仅针对以 I/O 重定向运算符所打开的文件或管道，还有 getline、print 或 printf。

system() 函数提供了简单删除脚本临时性文件的方式：

```
system("rm -f " tmpfile)
```

传递给 system() 的命令可包含数行：

```
system("cat <<EOFILE\nuno\ndos\ntres\nEOFILE")
```

它产生的输出和从嵌入文件复制到标准输出一样：

```
uno
dos
tres
```

由于每次调用 system() 都会起始一个全新的 Shell，因此没有简单的方式可以在分开的 system() 调用内的命令之间传递数据，除非通过中间文件。下面有个简单解决方案，将输出管道传送到 Shell，以送出多个命令：

```
Shell = "/usr/local/bin/ksh"
print "export INPUTFILE=/var/tmp/myfile.in" | Shell
print "export OUTPUTFILE=/var/tmp/myfile.out" | Shell
print "env | grep PUTFILE" | Shell
close(Shell)
```

这种方法还提供一个功能是：你可以选择Shell，不过缺点是无法在所有平台上都能取回退出状态值。

9.8 用户定义函数

谈到这里，我们所提过的awk语句已足够编写任何数据处理程序了。由于站在人类的角度，很难了解大型程序块，因此我们需要将这类的块切割成更易于管理的小数据块。绝大多数程序语言都能通过各式各样的函数、方法、模块、包及子例程，完成此功能。为达到简化，awk只提供了函数。就和C一样，awk函数也可选择性地返回标量值。只有该函数的文件或是代码可以清楚地说明调用者是否可以如期地得到返回值。

函数可定义在程序顶层的任何位置：成对的模式/操作组之前、之间、之后。在单一文件的程序里，惯例是将所有函数放在成对的模式/操作码之后，且让它们依字母顺序排列，这对人类而言，读起来会很方便，不过对awk而言并没有任何特别之处。

函数定义如下：

```
function name(arg1, arg2, ..., argn)
{
    statement(s)
}
```

指定的参数在函数体中用来当作局部变量，它们会隐藏任何相同名称的全局性变量。函数也可用于程序它处，调用的形式为：

```
name(expr1, expr2, ..., exprn)                忽略任何的返回值

result = name(expr1, expr2, ..., exprn)       将返回值存储到result中
```

在每个调用点上的表达式，都提供初始值给函数参数型变量。以圆括号框起来的参数，必须紧接于函数名称之后，中间没有任何空白。

对标量参数所做的变动，调用者无从得知，不过对数组的变动就可看见了。换句话说，标量为传值（by vaule），而数组则为传引用（by reference）：这对C语言也是这样。

函数体里的return *expression*语句会终止主体的执行，并将*expression*的值与控制权传给调用点。如果*expression*省略，则返回值由实现期定义。我们测试过的所有

系统，返回的不是数字零就是空字符串。POSIX则未对漏失return语句或值时的议题给出说明。

所有用于函数体且未出现在参数列表里的变量，都被视为全局性（global）的。awk允许在被调用函数中的参数比函数定义里所声明的参数还要少，额外的参数会被视为局部（local）变量。这类变量一般都用得到，所以惯例上是将它们列在函数参数列表里，并在字首前置一些额外的空白，如例9-2所示。这个额外参数就如同awk里的其他变量一样，在函数内容中会初始化为空字符串。

例 9-2：在数组中查找一值

```
function find_key(array, value,         key)
{
    # 查找array[]以寻找value，并返回array[key] == value
    # 的key，如果找不到该值，则返回""

    for (key in array)
        if (array[key] == value)
            return key
    return ""
}
```

如果无法成功地将局部变量列为额外的函数参数，则在调用程序使用到变量时，会很难找到bug。gawk提供了--dump-variables选项协助这一检查操作。

awk 就像大部分的程序语言：其函数也能调用自已，这就是大家所知道的递归（recursion）。显然这时候，程序设计就必须准备好什么时候该结束递归：一般的做法是，在每个连续性的调用上，都让工作变得越来越少，这么一来到了某个节点就没有再进一步递归的必要了。例9-3展现的是一个著名的例子，其基础的数字理论是由著名的希腊数学家欧几里德所提出的方法，寻找两个整数的最大公分母。

例 9-3：欧几里德的最大公分母算法

```
function gcd(x, y,         r)
{
    # 返回整数 x 与 y 的最大公分母

    x = int(x)
    y = int(y)
    # print x, y
    r = x % y
    return (r == 0) ? y : gcd(y, r)
}
```

如果我们增加这个操作

```
    { g = gcd($1, $2); print "gcd(" $1 ", " $2 ") =", g }
```

到例9-3的代码中，拿掉print语句的注释，并从文件执行它，便能看到循环的运行方式：

```
$ echo 25770 30972 | awk -f gcd.awk
25770 30972
30972 25770
25770 5202
5202 4962
4962 240
240 162
162 78
78 6
gcd(25770, 30972) = 6
```

欧几里德算法所采用的步骤相对较少，所以，没有awk里调用堆栈（call stack）溢出的危险，调用堆栈用于保存嵌套函数调用历史的数据。然而，并非总是这个情况，还有一特殊的嵌套函数，是由德国数学家Wilhelm Ackermann（注3）在1926年所发现，其值与递归深度的成长速度都比指数来得快很多。它可以用awk的代码定义，见例9-4。

例9-4：Ackermann之worse-than-exponential函数

```
function ack(a, b)
{
    N++                                # 计算递归深度
    if (a == 0)
        return (b + 1)
    else if (b == 0)
        return (ack(a - 1, 1))
    else
        return (ack(a - 1, ack(a, b - 1)))
}
```

如果我们将测试操作当作它的参数：

```
{ N = 0; print "ack(" $1 ", " $2 ") = ", ack($1, $2), "[" N " calls]" }
```

然后在测试文件中执行它，会发现：

```
$ echo 2 2 | awk -f ackermann.awk
ack(2, 2) =  7 [27 calls]

$ echo 3 3 | awk -f ackermann.awk
ack(3, 3) =  61 [2432 calls]

$ echo 3 4 | awk -f ackermann.awk
ack(3, 4) =  125 [10307 calls]
```

注3：见*http://mathworld.wolfram.com/AckermannFunction.html*可了解Ackermann函数的背景与历史信息。

```
$ echo 3 8 | awk -f ackermann.awk
ack(3, 8) =  2045 [2785999 calls]
```

ack(4, 4) 是完全无法计算的。

9.9 字符串函数

在 9.3.2 节里我们介绍过 length(*string*) 函数，用来返回字符串 *string* 的长度。其他常见的字符串运算，则包括有连接、数据格式化、字母大小写转换、匹配、查找、分割、字符串替换，以及子字符串提取。

9.9.1 子字符串提取

提取子字符串的函数：substr(*string*, *start*, *len*)，会返回一份由 *string* 的 *start* 字符开始，共 *len* 个字符长度的子字符串副本。字符的位置，从 1 开始编号：substr("abcde", 2, 3) 将返回 "bcd"。*len* 参数可省略，省略时，则默认为 length(*string*) - *start* + 1，选出字符串的剩余部分。

substr() 里的参数超出范围时不是一个错误，但是结果会视实际情况而定。例如 nawk 与 gawk 计算 substr("ABC", -3, 2) 的结果为 "AB"，而 mawk 产生的是空字符串 ""。如果为 substr("ABC", 4, 2) 与 substr("ABC", 1, 0) 则上述三者所得到的结果都为空字符串。gawk 的 --lint 选项可诊断出 substr() 调用里超出范围的参数。

9.9.2 字母大小写转换

有些字母表将大写与小写视为不同格式，在字符串查找与匹配中，通常会要求忽略字母大小写。awk 提供了两种函数来做这件事：tolower(*string*) 会返回将所有字母改为同义的小写的 *string* 副本，而 toupper(*string*) 则返回被改为大写字母的 string 副本。所以 tolower("aBcDeF123") 返回 "abcdef123"，toupper("aBcDeF123") 返回 "ABCDEF123"。这些功能在 ASCII 字母下可运行无误，但无法确切地转换重音字母。它们也无法正确地处理罕见字母，例如德文小写的 ß 均发 s 的尖锐音)，其大写形式为两个字母 SS。

9.9.3 字符串查找

index(*string*, *find*) 查找 *string* 里是否有字符串 *find*，然后返回 *string* 里 *find* 字符串的起始位置，如果在 *string* 里找不到 *find*，则返回 0。例如 index("abcdef", "de") 会返回 4。在 9.9.2 节里会告诉你，例如 index(tolower(*string*),

tolower(*find*))可以在查找字符串时忽略大小写。由于整个程序里，有时会需要区分大小写，所以gawk提供了一个好用的扩展：设置内建变量IGNORECASE为非零值，以忽略在字符串匹配、查找以及比较时字母的大小写。

index()会寻找第一个出现的子字符串，不过有时你要的是最后一个。并没有标准的函数可以做这件事，不过可以自己写一个，很简单，见例9-5。

例9-5：反向的字符串查找

```
function rindex(string, find,        k, ns, nf)
{
    # 返回 string 里最后一个出现的 find 的索引
    # 如果找不到，则返回 0

    ns = length(string)
    nf = length(find)
    for (k = ns + 1 - nf; k >= 1; k--)
        if (substr(string, k, nf) == find)
            return k
    return 0
}
```

循环由k值开始，对齐字符串string与find的结尾，从string中提取出与find等长的子字符串，与find比较。如果匹配，则k就是你要的最后一个出现的索引，然后函数返回该值。否则，我们再退回一个字符，直到k退回到string的开头才终止循环。如果一直退到起始处都无法匹配成功，则表示在string里找不到find，我们就返回索引为0。

9.9.4 字符串匹配

match(*string*, *regexp*)将*string*与正则表达式*regexp*匹配，如果匹配，则返回匹配*string*的索引，不匹配，则返回回0。这种方式提供了比表达式(*string* ~ *regexp*)还多的信息，后者只能得到计算值1或0。另外match()也具有一个有用的副作用：它会将全局变量RSTART设为在*string*中要开始匹配的索引值，而将RLENGTH设为要匹配的长度。而匹配子字符串则以substr(*string*, RSTART, RLENGTH)表示。

9.9.5 字符串替换

awk在字符串替换功能上，提供两个函数：sub(*regexp*, *replacement*, *target*)与gsub(*regexp*, *replacement*, *target*)。sub()将*target*与正则表达式*regexp*进行匹配，将最左边最长的匹配部分替换为字符串*replacement*。gsub()的运行则有点类似，不过它会替换所有匹配的字符串（前置g表示*global*全局之意）。这两种函数

都返回替换的数目。如省略第三个参数，则其默认值为当前的记录$0。这两个函数是不常用的，因为它们会修改标量参数：因此，它们无法以awk语言本身来编写。举例来说，支票开具记录软件可能会使用gsub(/[^$-0-9.,]/, "*", amount)将所有不能公开的金额全替换为星号。

在sub(*regexp*, *replacement*, *target*)或gsub(*regexp*, *replacement*, *target*)的调用里，每个*replacement*里的字符&都会被替换为*target*中与*regexp*匹配的文本。使用\&可关闭这一功能，而且请记得如果你要在引号字符串里使用它时，以双反斜杠转义它。例如gsub(/[aeiouyAEIOUY]/, "&&")令所有当前记录$0里的元音字母乘以两倍，而gsub(/[aeiouyAEIOUY]/, "\\&\\&")则是将所有的元音字母替换为一对&符号。

gawk提供了一个通用性的函数：gensub()，详细用法可参见gawk(1)使用手册。

要让数据减少，替换通常比索引与子字符串运算好用。可以想一下，从具有如下文本文件内的一个赋值中，提取字符串值的问题，如下所示：

```
composer =       "P. D. Q. Bach"
```

如果以替换的方式，则我们可以用：

```
value = $0
sub(/^ *[a-z]+ *= *"/, "", value)
sub(/" *$/, "", value)
```

不过以索引方式，则会像这样：

```
start = index($0, "\"") + 1
end = start - 1 + index(substr($0, start), "\"")
value = substr($0, start, end - start)
```

我们得更小心地计算字符，也无法准确地匹配数据模式，且还得建立两个子字符串。

9.9.6 字符串分割

awk针对当前输入记录$0自动提供了方便的分割为$1、$2、…、$NF，也可以函数来做：split(*string*, *array*, *regexp*)将*string*切割为片段，并存储到*array*里的连续元素。在数组里，片段放置在匹配正则表达式*regexp*的子字符串之间。如果*regexp*省略，则使用内建字段分隔字符FS的当前默认值。函数会返回*array*里的元素数量。例9-6为示范split()的用法。

例 9-6：分割字段程序的测试

```
{
    print "\nField separator = FS = \"" FS "\""
    n = split($0, parts)
    for (k = 1; k <= n; k++)
        print "parts[" k "] = \"" parts[k] "\""

    print "\nField separator = \"[ ]\""
    n = split($0, parts, "[ ]")
    for (k = 1; k <= n; k++)
        print "parts[" k "] = \"" parts[k] "\""

    print "\nField separator = \":\""
    n = split($0, parts, ":")
    for (k = 1; k <= n; k++)
        print "parts[" k "] = \"" parts[k] "\""
    print ""
}
```

如果我们将例9-6的程序放进文件里，而且以互动模式执行它，即可了解split()的运行：

```
$ awk -f split.awk
  Harold  and Maude

Field separator = FS = " "
parts[1] = "Harold"
parts[2] = "and"
parts[3] = "Maude"

Field separator = "[ ]"
parts[1] = ""
parts[2] = ""
parts[3] = "Harold"
parts[4] = ""
parts[5] = "and"
parts[6] = "Maude"

Field separator = :
parts[1] = "  Harold  and Maude"

root:x:0:1:The Omnipotent Super User:/root:/sbin/sh

Field separator = FS = " "
parts[1] = "root:x:0:1:The"
parts[2] = "Omnipotent"
parts[3] = "Super"
parts[4] = "User:/root:/sbin/sh"

Field separator = "[ ]"
parts[1] = "root:x:0:1:The"
parts[2] = "Omnipotent"
```

```
parts[3] = "Super"
parts[4] = "User:/root:/sbin/sh"

Field separator = ":"
parts[1] = "root"
parts[2] = "x"
parts[3] = "0"
parts[4] = "1"
parts[5] = "The Omnipotent Super User"
parts[6] = "/root"
parts[7] = "/sbin/sh"
```

请特别留意默认字段分隔字符值" "与"[ ]"的差异：前者会忽略前置与结尾的空白，并于运行时将空白（whitespace）视为一个单独空格（single space），后者则正好匹配一个空格。对绝大多数文本处理应用程序而言，第一种行为模式就已经满足功能上的需求了。

以冒号为字段分隔字符的例子显示出：当字段分隔字符不匹配时，则 `split()` 会产生单元素数组（one-element array），并展示切割传统 UNIX 管理文件 `/etc/passwd` 里的记录。

近期 awk 的实现提供更一般化的方式：`split(string, chars, "")`，将 *string* 分割为单字符元素放置到 `chars[1]`、`chars[2]`、…、`chars[length(string)]` 中。旧式的实现则要求使用下面这种较没有效率的方式：

```
n = length(string)
for (k = 1; k <= n; k++)
    chars[k] = substr(string, k, 1)
```

调用 `split("", array)` 可删除 *array* 里的所有元素，这个方法比使用循环进行数组元素删除还快：

```
for (key in array)
    delete array[key]
```

当你的 awk 实现不支持 `delete array` 时可以使用。

如果你需要在 awk 里通过多下标数组执行重复操作，则 `split()` 就是不可或缺的函数了。这里是它的范例：

```
for (triple in maildrop)
{
    split(triple, parts, SUBSEP)
    house_number = parts[1]
    street = parts[2]
    postal_code = parts[3]
    ...
}
```

9.9.7 字符串重建

awk并无标准内建函数可执行split()的反置处理，不过要写一个也很简单，见例9-7。join()可确保参数数组不会被引用到，除非索引是在范围之内。否则，一个具有数组长度为0的调用可能会建立array[1]，而修改了调用者的数组。插入的字段分隔字符为普通字符串，而非正则表达式，所以针对传递给split()的一般正则表达式，join()不会重建精确的原始字符串。

例9-7：将数组元素组合为字符串

```
function join(array, n, fs,         k, s)
{
    # 重新组合 array[1]…array[n] 为一个字符串
    # 并以 fs 分隔数组元素

    if (n >= 1)
    {
        s = array[1]
        for (k = 2; k <= n; k++)
            s = s fs array[k]
    }
    return (s)
}
```

9.9.8 字符串格式化

最后一个与字符串相关的函数是在用户控制下格式化数字与字符串：sprintf(*format*, *expression1*, *expression2*, ...)，它会返回已格式化的字符串作为其函数值。printf()的运行方式也是这样，只不过它会在标准输出或重定向的文件上显示格式化后的字符串，而不是返回其函数值。较新的程序语言以更强大的格式化函数来取代格式控制字符串，但相对而言让代码变得很冗长。按照传统的文本处理应用来说，sprintf()与printf()几乎就够用了。

printf()与sprintf()的格式字符串有点类似在Shell里的printf命令，详见7.4节。我们将awk的格式项目概括于表9-5。这些项目每一个都可以用相同字段宽度、精度以及第7章讨论的标志修改符来增加。

%i、%u与%X并非1987年语言重新设计时的一部分，不过现代的实现都支持它们。尽管与Shell的printf命令里很相似，但awk的%c对整数参数方面的处理是不同的，且使用%u的输出时，对负数参数的处理上也有差异，这是由于Shell与awk在算术上的不同所导致的。

表 9-5：printf 与 sprintf 格式描述符

项目	说明
%c	ASCII 字符。显示相对应于字符串参数的第一个字符，或是在主机字符集里，相对应于该整数参数的编号的字符，通常是 256 的余数。
%d, %i	十进制整数。
%e	浮点格式（[-]*d.precision*e[+-]*dd*）。
%f	浮点格式（[-]*ddd.precision*）。
%g	%e 或 %f 的转换，因为删除结尾的 0，所以较短。
%o	无符号八进制值。
%s	字符串。
%u	不带正负号的值。awk 数字是浮点数值：小的负值整数会以大的正值输出，因为符号字节被解释为一个数据位。
%x	不带正负号的十六进制数字。字母 a-f 表示 10 到 15。
%X	不带正负号的十六进制数字。字母 A-F 表示 10 到 15。
%%	字面上的 %。

大部分的格式项目都是直觉易懂的。不过我们还是得特别留意二进制浮点数值转换为十进制字符串时能达到的精度，反向运算也然，都可能出现难以解决的大问题。比较好的解决方案只有在 1990 年发现的那个，而它需要极高的精度。awk 实现一般是使用底层的C函数库，进行 sprintf()格式项目所需的转换，虽然函数库的品质一直在改善，但仍有一些平台的浮点转换精度存在不足。再者，浮点运行硬件的不同与命令计算顺序的差异，意味着只要硬件架构稍有不同，几乎来自任何程序语言所产生的浮点运算结果就会有些许不同。

当浮点数出现在 print 语句里时，awk 会根据内建变量 OFMT 的值格式化它们，OFMT 的默认值为 "%.6g"。如果有必要，可重新定义。

类似地，当浮点数转换为连续字符串时，awk 会根据另一个内建变量 CONVFMT（注 4）的值进行格式化。CONVFMT 的默认值也为 "%.6g"。

例 9-8 测试程序所产生的输出，有点类似近期 Sun Solaris 的 SPARC 系统提供的 nawk 版本所产生的结果：

注 4：最初，OFMT 做的是输出与字符串的转换，但 POSIX 提出 CONVFMT 后，将它们两个的用途做了区分。大部分的实现两者都支持，但 SGI IRIX 与 Sun Solaris /usr/bin/nawk 不支持 CONVFMT。

```
$ nawk -f ofmt.awk
[ 1] OFMT = "%.6g"      123.457
[ 2] OFMT = "%d"        123
[ 3] OFMT = "%e"        1.234568e+02
[ 4] OFMT = "%f"        123.456789
[ 5] OFMT = "%g"        123.457
[ 6] OFMT = "%25.16e"      1.2345678901234568e+02
[ 7] OFMT = "%25.16f"        123.4567890123456806
[ 8] OFMT = "%25.16g"           123.4567890123457
[ 9] OFMT = "%25d"                            123
[10] OFMT = "%.25d"     0000000000000000000000123
[11] OFMT = "%25d"                     2147483647
[12] OFMT = "%25d"                     2147483647    预期的 2147483648
[13] OFMT = "%25d"                     2147483647    预期的 9007199254740991
[14] OFMT = "%25.0f"             9007199254740991
```

显然，尽管在浮点值里可以表示到53位的精确度，但在这个平台上的nawk，在%d格式的限定下，只支持到32位的整数。相同的nawk版本，在不同架构下执行，便产生略微不同的结果。例9-8为ofmt.awk的源代码。

例 9-8：测试 OFMT 的效果

```
BEGIN {
    test( 1, OFMT,      123.4567890123456789)
    test( 2, "%d",      123.4567890123456789)
    test( 3, "%e",      123.4567890123456789)
    test( 4, "%f",      123.4567890123456789)
    test( 5, "%g",      123.4567890123456789)
    test( 6, "%25.16e", 123.4567890123456789)
    test( 7, "%25.16f", 123.4567890123456789)
    test( 8, "%25.16g", 123.4567890123456789)
    test( 9, "%25d",    123.4567890123456789)
    test(10, "%.25d",   123.4567890123456789)
    test(11, "%25d",    2^31 - 1)
    test(12, "%25d",    2^31)
    test(13, "%25d",    2^52 + (2^52 - 1))
    test(14, "%25.0f",  2^52 + (2^52 - 1))
}

function test(n,fmt,value,     save_fmt)
{
    save_fmt = OFMT
    OFMT = fmt
    printf("[%2d] OFMT = \"%s\"\t", n, OFMT)
    print value
    OFMT = save_fmt
}
```

我们发现，对于在不同的awk实现，这个测试的输出会有完全不同的结果，甚至相同程序在不同发布版本下也可能出现不同的结果。例如，以gawk执行，我们会得到：

```
$ gawk -f ofmt.awk
...
```

```
[11] OFMT = "%25d"       2147483647                               预期向右对齐的结果
...
[13] OFMT = "%25d"                            9.0072e+15          预期的 9007199254740991
...
```

1987年出版的一本awk书里，对OFMT定义了默认值，不过这是非正式的语言定义，除此之外该书并未提及其他的值。有可能各实现在认知上有所不同，POSIX宣称，如果OFMT不是浮点格式规格的话，则转换结果是未指定的，因此在这里的gawk行为是被允许的。

使用mawk，我们发现：

```
$ mawk -f ofmt.awk
...
[ 2] OFMT = "%d"         1079958844                               预期 123
...
[ 9] OFMT = "%25d"                            1079958844          预期 123
[10] OFMT = "%.25d"      0000000000000001079958844                预期 00…00123
[11] OFMT = "%25d"       2147483647                               预期结果向右对齐
[12] OFMT = "%25d"                            1105199104          预期 2147483648
[13] OFMT = "%25d"                            1128267775          预期 9007199254740991
...
```

显然这里在处理格式为%d与%i的大型数字时会有不一致的特殊处理手法。幸好，改用%.0f格式，便能从所有awk实现中，取得正确的输出。

9.10 数值函数

awk提供的基础数值函数参见表9-6。大部分都是在很多程序语言上常见的，且其精度由底层本地数学函数库的品质而定。

表9-6：基础数值函数

函数	说明
atan2(*y*, *x*)	返回 *y*/*x* 的反正切，值介于 -π 与 +π 之间。
cos(*x*)	返回 *x* 的余弦值（以弧度（radians）计算），该值介于 −1 与 +1 之间。
exp(*x*)	返回 *x* 的指数，*ex*。
int(*x*)	返回 *x* 的整数部分，截去前置的 0。
log(*x*)	返回 *x* 的自然对数。
rand()	返回平均分布的虚拟随机 *r*，0 ≤ *r*<1。
sin(*x*)	返回 *x* 的正弦值（以弧度（radians）计算），该值介于 −1 与 +1 之间。

表 9-6：基础数值函数（续）

函数	说明
sqrt(*x*)	返回 *x* 的平方。
srand(*x*)	设置虚拟随机产生器的种子为 *x*，并返回正确的种子。如果省略 *x*，则使用当前时间（以秒计）。如果 srand() 未被调用，则 awk 在每次执行时会从相同的默认种子开始；mawk 则不会。

虚拟随机数产生器函数rand()与srand()是各种awk实现中程序库函数里差异最大的，因为有些实现使用本地的系统程序库函数，而非自有代码，且虚拟随机产生算法与精度也有所不同。大部分产生这类数字的算法，是经过一组有限的序列而没有重复的步骤，且在产生器周期（period）中被调用的一些步骤之后，该序列最终会重复它自己。程序库文件有时并没有说清楚，单元区间端点0.0与1.0是否被包含在rand()的范围内，或者周期为多少。

产生器的结果区间端点的不明确性使得程序很难写。假定想产生介于 0 到 100（含）的虚拟随机整数，如果使用简单表达式 int(rand()*100)，则当 rand() 不返回 1.0 时，你根本不可能得到 100 的值，但就算返回该值，得到 100 的次数还是要比 0 与 100 间的其他整数少得多，因为在产生器周期里，它只被产生一次。如果你想，那就把 100 改成 101 就好了，这也不可行，因为在某些系统下，有可能得到 101 这种超出范围的结果。

例 9-9 的 irand() 函数提供了产生虚拟随机整数的较好方法。irand() 强制整数端点，且如果要求的范围为空或无效时，则返回一个端点值。否则，irand() 取样大于区间宽度的整数，把它增加到 low，并且如果结果超出范围，则重试。现在，rand() 是否会返回 1.0 已不是问题，且来自 irand() 的返回值，将和 rand() 值一样，平均分布。

例 9-9：产生虚拟随机整数

```
function irand(low, high,        n)
{
    # 返回虚拟随机整数 n，使得 low <= n <= high

    # 确保整数端点
    low = int(low)
    high = int(high)

    # 参数顺序的健康检查
    if (low >= high)
        return (low)

    #  在要求的区间里寻找值
    do
        n = low + int(rand() * (high + 1 - low))
    while ((n < low) || (high < n))
```

```
    return (n)
}
```

如果程序未调用 srand(*x*)，则 gawk 与 nawk 会在每次执行时都使用相同的初始种子；mawk 则不是这样。通过调用 srand() 将当前的时间作为种子，使得在每次执行时取得不同的序列，这是合理的，但这是假设时间的准确度足够。遗憾的是，虽然机器的速度与日俱增，但大部分用于现行 awk 实现里的时钟其时间精确度只是秒而已，所以极可能一连串的模拟执行都在相同的时刻内完成。解决方法是：避免在每次执行中调用 srand() 一次以上，并在每次执行之间插入至少一秒的延迟：

```
$ for k in 1 2 3 4 5
> do
>     awk 'BEGIN {
>                 srand()
>                 for (k = 1; k <= 5; k++)
>                     printf("%.5f ", rand())
>                 print ""
>            }'
>     sleep 1
> done
0.29994 0.00751 0.57271 0.26084 0.76031
0.81381 0.52809 0.57656 0.12040 0.60115
0.32768 0.04868 0.58040 0.98001 0.44200
0.84155 0.56929 0.58422 0.83956 0.28288
0.35539 0.08985 0.58806 0.69915 0.12372
```

没有 sleep 1 语句的话，输出行很有可能会完全一样。

9.11 小结

我们在本章所展现的 awk 子集，已经能做相当多的文本处理工作。当你了解 awk 的命令行，并知道它如何自动处理输入文件，那么写程序的工作就简化到只需要标明记录的选定与其相对应的操作。这类极简单的数据驱动程序语言相当具有效率。相比之下，大多数传统的程序语言，都必须苦思如何使用循环处理一连串输入的文件，并针对每个文件处理打开文件、读取、选定以及处理记录直到文件结尾，还有最后的关闭文件，然后把这些写成一长串的例程。

当你看到使用 awk 处理记录与字段有多简单时，对数据处理的看法就会大大改变。你可以开始将大型的任务，分割为更小、更易于管理的工作。例如，当你面临必须处理一个极复杂的二进制文件时，可能是用于数据库的文件，或是字体、图片、幻灯片、电子表格、排版软件及字处理器等文件，你可以设计或直接找一组可以将二进制格式转换为适当标记的纯文本格式的工具，然后再在 awk 或其他脚本语言里写一个小小的过滤程序，处理文本的显示。

第10章

文件处理

本章要讨论的是在处理文件时常见的一些命令，例如：列出文件、修改它们的时间戳、建立临时性文件、在目录层级中寻找文件、将命令应用到文件列表、计算文件系统空间的使用量以及比较文件。

10.1 列出文件

echo命令提供简单的方式列出匹配模式的文件：

```
$ echo /bin/*sh                              显示/bin下的Shell
/bin/ash /bin/bash /bin/bsh /bin/csh /bin/ksh /bin/sh /bin/tcsh /bin/zsh
```

Shell将通配字符模式替换为匹配的文件列表，而且echo以空格区分文件列表，在单一行上显示它们。不过echo不会更进一步解释它的参数，因此与文件系统里的文件也没有任何关联。

ls命令则比echo能作更多的处理，因为它知道自已的参数应该是文件。未提供命令行选项时，ls只会验证其参数是否存在，并显示它们，如果输出并非终端，则以一行一个的方式显示，如果是终端，则为多栏显示模式。马上看看这三种有何不同：

```
$ ls /bin/*sh | cat                          在输出管道里显示 Shell
/bin/ash
/bin/bash
/bin/bsh
/bin/csh
/bin/ksh
/bin/sh
/bin/tcsh
/bin/zsh
```

ls

语法

```
ls [ options ] [ file(s) ]
```

用途

列出文件目录的内容。

主要选项

-1

数字1。强制为单栏输出。在交互式模式下，ls一般会以适于当前窗口的最小宽度，使用多个列。

-a

显示所有文件，包括隐藏文件（文件名以点号起始的文件）。

-d

显示与目录本身相关的信息，而非它们包含的文件的信息。

-F

使用特殊结尾字符，标记特定的文件类型。

-g

仅适用于组：省略所有者名称（隐含-l，小写L选项）。

-i

列出inode编号。

-L

紧接着符号性连接，列出它们指向的文件。

-l

小写的L。以冗长形式列出，带有类型、权限保护、所有者、组、字节计数、最后修改时间和文件名。

-r

倒置默认的排序顺序。

-R

递归列出，下延进入每个子目录。

-S

按照由大到小的文件大小计数排序。仅GNU版本支持。

-s

以块（与系统有关）为单位，列出文件的大小。

ls（续）

-t

按照最后修改时间戳排序。

--full-time

显示完整的时间戳。仅GNU版本支持。

行为模式

ls通常只显示文件名称：如须取得与文件属性相关的信息，必须提供额外选项。文件是以辞典编排的顺序排序，不过可通过-S或-t选项改变它。排序的顺序是按照系统环境语言（locale）而定。

警告

许多ls实例都提供比这里介绍还要多的选项；请参阅你本地系统里的使用手册，了解更多信息。

```
$ ls /bin/*sh                                以80个字符宽度的终端窗口，显示Shell
/bin/ash  /bin/bash  /bin/bsh  /bin/csh  /bin/ksh  /bin/sh  /bin/tcsh
/bin/zsh

$ ls /bin/*sh                                以40个字符宽度的终端窗口，显示Shell
/bin/ash   /bin/csh  /bin/tcsh
/bin/bash  /bin/ksh  /bin/zsh
/bin/bsh   /bin/sh
```

为了终端输出时，ls会使用刚好适合的多栏，将数据依栏加以排列。这只是为了人们方便检查；如果你要单栏输出到终端，可使用ls -1（数字1）强制执行。另外，处理ls的管道输出的程序，可预期得到一个文件名一行的模式。

在BSD、GNU/Linux、Mac OS X与OSF/1的系统上，ls会将文件名里无法打印的字符，在终端输出时转换为问号，但报告文件名到非终端输出则不做改变。来看看这个特殊名称one\ntwo，\n为换行字符。这里是GNU ls的处理方式：

```
$ ls one*two                    列出特定的文件名
one?two

$ ls one*two | od -a -b         显示真实的文件名
0000000   o   n   e  nl   t   w   o  nl
        157 156 145 012 164 167 157 012
0000010
```

八进制输出工具程序od会显示真正的文件名：第一个报告换行字符为名称的一部分，第二个则结束输出行。下游的程序会看到两个明显分开的名称；在稍后的10.4.3节里会说明如何解决这样的错乱。

不同于echo的是：ls要求它的文件参数要存在，而且如果它们不存在的话，则会出现抱怨：

```
$ ls this-file-does-not-exist                试图列出一个不存在的文件
ls: this-file-does-not-exist: No such file or directory

$ echo $?                                    显示ls的离开码
1
```

没有参数时，echo只会显示一个空行，但ls会列出当前目录的内容。我们先产生一个含有三个文件的目录，以便讲解其行为模式：

```
$ mkdir sample                               建立新目录
$ cd sample                                  切换到此新目录
$ touch one two three                        建立空白文件
```

然后应用echo与ls到它的内容：

```
$ echo *                                     回应匹配的文件
one three two

$ ls *                                       列出匹配的文件
one  three  two

$ echo                                       不带参数的echo
                                             这个输出行是空的

$ ls                                         列出当前的目录
one  three  two
```

以一个点号为开头的文件名，在正规Shell模式匹配中会被隐藏。我们来看看在一个含有三个隐藏文件的子目录中，这类文件是如何被处理的，有何不同：

```
$ mkdir hidden                               建立新目录
$ cd hidden                                  切换到该目录
$ touch .uno .dos .tres                      建立三个隐藏的空文件
```

接下来尝试显示它的内容：

```
$ echo *                                     回应匹配的文件
*                                            未有匹配者

$ ls                                         列出非隐藏文件
                                             这个输出行是空的

$ ls *                                       列出匹配的文件
ls: *: No such file or directory
```

当没有匹配模式的文件时，Shell会将模式视为参数：在这里echo看到星号并打印它，而ls则试图寻找名为*的文件，然后报告寻找失败。

现在，如果我们提供匹配前置点号的模式，可再进一步了解它们的差异：

```
$ echo .*                                    回应隐藏文件
. .. .dos .tres .uno

$ ls .*                                      列出隐藏文件
.dos  .tres  .uno

.:
..:
hidden  one  three  two
```

UNIX 目录总是包含特殊实例 ..（父目录）以及 .（当前目录），且 Shell 会传递所有的匹配给这两个程序。echo 只报告它们，但 ls 会做更多的事：当命令行参数为目录时，它会列出该目录的内容。在我们的例子里，这个列表会包含父目录的内容。

你可以显示目录本身的相关信息，而非其内容，只要使用 -d 选项即可：

```
$ ls -d .*                                   列出隐藏文件，但没有目录内容
.  ..  .dos  .tres  .uno

$ ls -d ../*                                 列出父文件，但没有目录内容
../hidden  ../one  ../three  ../two
```

由于你通常要的不是显示父目录，因此，ls 还提供了 -a 选项，提供打印当前目录里的所有文件，包含隐藏文件在内：

```
$ ls -a                                      列出所有文件，包括隐藏文件
.  ..  .dos  .tres  .uno
```

在此不会列出父目录的内容，因为没有参数指定它。

10.1.1 长的文件列出

由于 ls 知道它的参数是文件，所以可以进一步地报告相关细节，尤其是文件系统的一些 metadata，这个功能通常是以 -l（小写 L）选项完成：

```
$ ls -l /bin/*sh                    列出在 /bin 下的 Shell
-rwxr-xr-x  1 root root 110048 Jul 17  2002 /bin/ash
-rwxr-xr-x  1 root root 626124 Apr  9  2003 /bin/bash
lrwxrwxrwx  1 root root      3 May 11  2003 /bin/bsh -> ash
lrwxrwxrwx  1 root root      4 May 11  2003 /bin/csh -> tcsh
-rwxr-xr-x  1 root root 206642 Jun 28  2002 /bin/ksh
lrwxrwxrwx  1 root root      4 Aug  1  2003 /bin/sh -> bash
-rwxr-xr-x  1 root root 365432 Aug  8  2002 /bin/tcsh
-rwxr-xr-x  2 root root 463680 Jun 28  2002 /bin/zsh
```

虽然这种输出形式是常见的，但是额外的命令行选项，可对它的输出稍作修改。

每行上的首字符描述文件类型：- 为一般文件、d 为目录、l 为符号连接。

接下来的 9 个字符，则报告文件权限：针对每个用户、组以及除此外的其他人。r 表示读取、w 表示写入、x 表示执行，如果未提供权限则是 -。

第二栏包含连接计数：在这里，只有 /bin/zsh 拥有直接链接到另一个文件，但是还有其他的文件未显示于这里的输出，因为它们的名称与参数模式不匹配。

第三栏、第四栏报告文件所有者与所属组，第五栏则是以字节为单位的文件大小。

接下来的三栏是最后修改的时间戳。这里显示的是一直沿用下来的形式：月、日、年，表示六个月前的文件，其他的文件则是年的部分会被替换为时间（指六个月内的文件）：

```
$ ls -l /usr/local/bin/ksh                         列出最近的文件
-rwxrwxr-x  1 jones devel 879740 Feb 23 07:33 /usr/local/bin/ksh
```

不过，在现代的 ls 实例上，时间戳与 locale 相关，且使用较少的栏。这里是在 GNU/Linux 上测试的两种 ls 版本：

```
$ LC_TIME=de_CH /usr/local/bin/ls -l /bin/tcsh  列出 locale 为 Swiss-German
                                                的时间戳
-rwxr-xr-x  1 root root 365432 2002-08-08 02:34 /bin/tcsh

$ LC_TIME=fr_BE /bin/ls -l /bin/tcsh            列出 locale 为 Belgian-French
                                                的时间戳
-rwxr-xr-x    1 root     root       365432 août  8  2002 /bin/tcsh
```

尽管时间戳应该已经国际化，但这个系统在 English 原型下，报告错误的 French 时间 le 8 août 2002。

GNU 版本允许显示完整的时间精准度；下面的例子是来自 SGI IRIX 系统，显示一百万分之一秒精准度：

```
$ /usr/local/bin/ls -l --full-time /bin/tcsh      高精准度的时间戳
-r-xr-xr-x  1 root sys 425756 1999-11-04 13:08:46.282188000 -0700 /bin/tcsh
```

前面的 ls 命令说明栏里，呈现 ls 实例的一些通用选项，但其实还有更多：GNU 的版本就有将近 40 种选项！你将会时常用到 ls，所以偶尔重新详读它的手册页，更新你的记忆，绝对很有帮助。如果你要做的是具可移植性的 Shell 脚本，请限制自己使用较通用的选项，并设置环境变量 LC_TIME，可减少因 locale 所产生的变异。

10.1.2 列出文件的 meta 数据

当计算机以精简的二进制形式存储数据时，针对相同的数据，能够以更详尽的形式提供其数据内容，可方便人们或简单的计算机程序阅读，这是非常有用的。我们在本书中已

使用八进制输出工具od多次，它可以将无法打印的数据流字节转换为文字，而我们也将于13.7节探讨特殊文件系统/proc，它可以让内部核心程序里的数据更易于访问。

奇怪的是，文件系统里的meta数据，通过POSIX标准下的fstat()、lstat()与stat()函数库调用，已长期被C程序设计员使用，但在Shell及脚本语言里，除了ls命令所提供的有限形式以外，很难被程序设计员访问。

20世纪90年代末期，SGI IRIX提出stat命令。在2001年左右，为BSD系统及GNU *coreutils*包所写的stat独立实例也出现了。不幸的是，这三个程序的输出格式完全不同，见附录B的说明。它们各自拥有为数众多的命令行选项，提供更多对输出何种数据以及使用何种格式的控制。GNU版本是唯一构建在各种UNIX版本之上的，所以如果你在它之上进行标准化，便能在你的本地Shell脚本内好好利用它的功能。

10.2 使用touch更新修改时间

我们已介绍过使用touch命令建立空文件。对于之前不存在的文件，下面提供的几种方式，都为完成相同目的：

```
cat /dev/null  > some-file                 复制空文件到some-file
printf ""      > some-file                 打印空字符串到some-file
cat /dev/null >> some-file                 附加空文件到some-file
printf ""     >> some-file                 附加空字符串到some-file
touch            some-file                 更新some-file的时间戳
```

不过，如果是文件已存在，前两个操作就会将文件大小删减到0，后面的三种事实上什么事也不做，只更新最后修改时间。说得更清楚些：比较安全的做法是使用touch，因为如果你的意思是>>却不小心输入为>时，就会毁了文件内容。

有时在Shell脚本里也会应用touch建立空文件：它们的存在与时间戳是有意义的，但它们的内容则否。最常见的例子是用于锁定文件，以指出程序已在执行中，不应启动第二个实例（instance）。另一种用途则为记录文件的时间戳，供日后与其他文件对照用。

touch默认（或使用-m选项）操作会改变文件的最后修改时间，不过你也可以使用-a选项改变文件的最后访问时间。时间部分，默认为使用当前时间，但你也可以搭配-t选项覆盖之，方式是加上[[CC]YY]MMDDhhmm[.SS]形式的参数，世纪、公元年和秒数是可选用的，月份则为01到12、日期范围为01到31，时区为当地时区。例如：

```
$ touch -t 197607040000.00 US-bicentennial     建立生日文件

$ ls -l US-bicentennial                         列出文件
-rw-rw-r--  1 jones devel 0 Jul  4  1976 US-bicentennial
```

touch 还提供 -r 选项，复制参照文件的时间戳：

```
$ touch -r US-bicentennial birthday          把时间戳复制到新的 birthday 文件

$ ls -l birthday                             列出新文件
-rw-rw-r--  1 jones devel 0 Jul  4  1976 birthday
```

旧系统上的touch命令并没有-r选项，不过所有现行版本都支持此功能，且POSIX也要求具有它。

以日期来看，UNIX时间戳（epoch）是从零开始，由1970/1/1 00:00:00 UTC（注1）算起。大部分现行系统都有一个带正负号32位的时间计数器，每一秒加1，且允许日期的表示往前推到1901年晚期，往后则到2038年；当计时器在2038年溢出时，它就会回到1901。幸好，一些近期的系统已经切换到64位计数器：即使以一百万分之一秒计算，它还是能扩展到五十万年以上！ 32位与64位计时器的时钟比较如下：

```
$ touch -t 178907140000.00 first-Bastille-day     为法国建国日建立一个文件
touch: invalid date format `178907140000.00'      32 位计时器显然是不足的

$ touch -t 178907140000.00 first-Bastille-day     再试试 64 位计时器

$ ls -l first-Bastille-day                        顺利运行，列出文件
-rw-rw-r--  1 jones devel 0 1789-07-14 00:00 first-Bastille-day
```

要在64位计时器时钟的系统上，以touch使用未来时间，仍是无法完成的，这是人为加的软件限制，因为人们错误地认为POSIX要求的世纪只需要两位数：

```
$ touch -t 999912312359.59 end-of-9999          可运行

$ ls -l end-of-9999                             列出文件
-rw-rw-r--  1 jones devel 0 9999-12-31 23:59 end-of-9999

$ touch -t 1000001010000.00 start-of-10000      失败
touch: invalid date format `1000001010000.00'
```

幸好，GNU 的 touch 提供另一种选项可用以规避 POSIX 的限制：

```
$ touch -d '10000000-01-01 00:00:00' start-of-10000000     进入下一个百万世纪!

$ ls -l start-of-10000000                                  列出文件
-rw-rw-r--  1 jones devel 0 10000000-01-01 00:00 start-of-10000000
```

10.3 临时性文件的建立与使用

虽然使用管道可以省去建立临时性文件的需求，不过有时临时性文件还是派得上用场的。UNIX不同于其他操作系统的地方就是：它没有那种将不再需要的文件设法神奇地删除

注1： UTC习惯称为GMT；请参考术语表中的 *Coordinated Universal Time*。

的想法。反倒是提供了两个特殊目录：/tmp与/var/tmp（旧系统为/usr/tmp），这些文件可如常被存储，当它们未被清理干净时也不会弄乱一般的目录。大部分系统上的/tmp都会在系统开机时清空，不过/var/tmp下的在重新开机时仍需存在，因为有些文字编辑程序,会将它们的备份文件存放在这里,从而系统毁损后可以用来恢复数据。

因为/tmp使用频繁，有些系统就会将它放在常驻内存型（memory-resident）的文件系统里，以便快速访问，如下面这个Sun Solaris系统里的例子：

```
$ df /tmp                          显示/tmp下的磁盘剩余空间
Filesystem          1K-blocks      Used Available Use% Mounted on
swap                 25199032    490168  24708864   2% /tmp
```

将文件系统放在替换空间（swap）区域里，表示它存在于内存中，直到内存资源被使用得剩很少时，部分信息才会写入替换空间。

注意： 临时性文件的目录是共享的资源，这也让它们成了拒绝服务（denial of service，DOS）攻击的目标。让其他工作填满整个文件系统（或替换空间），然后窥伺系统，或以其他用户身份删除文件。系统管理因此会监控这些目录的使用空间，然后执行cron作业，清掉旧文件。此外，这些目录通常都会设置粘连位（sticky permission bit），使得只有root与文件所有者可以删除它们。是否要设置文件权限以限制存储在这样的目录内的文件访问由你决定。Shell脚本应该都要使用umask命令（见附录B），或是先以touch建立必需的临时性文件，再执行chomd将之设置为适当权限。

为确保临时性文件会在任务完成时删除，编译语言的程序员可以先开启文件，再下达unlink()系统调用。这么做就会马上删除文件，但因为它仍在开启状态，所以仍可继续访问，直到文件关闭或工作结束为止，只要其中一个先发生即可。打开后解除连接（unlink-after-open）的技巧一般来说在非UNIX操作系统下是无法运行的，在加载于UNIX文件系统中目录上的外部文件系统也是这样，且在大多数脚本语言中是无法使用的。

注意： 很多系统上的/tmp与/var/tmp都是比较小的文件系统，且通常加载在独立于根（/）分区之外的个别分区（partition）上，所以当它被填满时，不会妨碍系统日志的记录。特别的一点是：这也意味着，你不能在这些目录下建立大型的临时性文件，供CD或DVD的文件系统映像使用。如果/tmp被填满了，你可能无法编辑程序，一直要等到你的系统管理员解决这个问题为止，除非，你的编译程序允许将临时性文件重定向到其他目录。

10.3.1 $$ 变量

共享的目录或同一程序的多个执行实例，都可能造成文件名冲突。在 Shell 脚本里的传统做法是使用进程 ID（见 13.2 节），可以在 Shell 变量 `$$` 中取得，构建成临时性文件名的一部分。要解决完整临时性文件名发生此问题的可能性，可使用环境变量覆盖目录名称，通常是 `TMPDIR`。另外，你应该使用 `trap` 命令，要求在工作完成时删除临时性文件，请见 13.3.2 节。因此，常见的 Shell 脚本起始如下：

```
umask 077                              删除用户以外其他人的所有访问权
TMPFILE=${TMPDIR-/tmp}/myprog.$$       产生临时性文件名
trap 'rm -f $TMPFILE' EXIT             完成时删除临时性文件
```

10.3.2 mktemp 程序

像 `/tmp/myprog.$$` 这样的文件名会有个问题：太好猜了！攻击者只需要在目标程序执行时列出目录几次，就可以找出它正在使用的是哪些临时性文件。通过预先建立适当的指定文件，攻击者可以让你的程序失败或读取伪造的数据，甚至重设文件权限，以便于他（攻击者）读取文件。

处理此类安全性议题时，文件名必须是不可预知的。BSD 与 GNU/Linux 系统都提供 `mktemp` 命令，供用户建立难以猜测的临时性文件名称。虽然底层的 `mktemp()` 函数库调用已由 POSIX 标准化，但 `mktemp` 命令却没有。如果你的系统没有 `mktemp`，我们建议你安装源自 OpenBSD 的可移植版本（注 2）。

`mktemp` 采用含有结尾 X 字符的文件名模板（可选用的），我们建议至少使用 12 个 X。程序会用从随机数字与进程ID所产生的文字或数字字符串来取代它们，所建立的文件名不允许组与其他人访问，然后将文件名打印在标准输出上。

注意： 这里是为什么我们建议你用 12 个或以上的 X 字符。容易可猜测的进程 ID 可能有 6、7 个，所以随机的字母数可能只有 5 个：那么有 52^5（3.8 亿）个随机字母字符串。然而，如果只是 10 个 X（这是 `mktemp` 的默认值，请参考手册页）及 7 个数字的 PID，那么只需要猜 14000 次。我们拿手边最快的机器，以 40 行 C 程序来测试这样的攻击，发现一百万种猜测可在 3 秒之内完成。

这里来看看 `mktemp` 的使用：

注 2：　可在 *ftp://ftp.mktemp.org/pub/mktemp/* 取得。

```
$ TMPFILE=`mktemp /tmp/myprog.XXXXXXXXXXXX` || exit 1   建立唯一的临时性文件

$ ls -l $TMPFILE                                         列出临时性文件
-rw-------   1 jones devel 0 Mar 17 07:30 /tmp/myprog.hJmNZbq25727
```

进程编号25727可从文件名结尾处看出，但副文件名的剩余部分就无法预测了。当临时性文件无法建立或没有mktemp可用时，条件式exit命令可确保马上终止程序并带有错误输出。

最新版的mktemp允许省略模板；它会使用默认的/tmp/tmp.XXXXXXXXXX。然而，较旧版本仍是需要模板，所以你的Shell脚本请避免使用这种省略方式。

警告： HP-UX的mktemp版本太弱了：它会忽略所有用户所提供的模板，然后以用户名称与进程ID重建一个好猜的临时性文件名。我们强烈建议你在HP-UX安装OpenBSD的版本。

为避免在程序里将目录名称直接编码（hardcode），可使用-t选项：让mktemp使用环境变量TMPDIR所指定的目录或/tmp。

-d选项要求建立临时性目录：

```
$ SCRATCHDIR=`mktemp -d -t myprog.XXXXXXXXXXXX` || exit 1    建立临时性目录

$ ls -lFd $SCRATCHDIR                                          列出目录本身
drwx------   2 jones devel 512 Mar 17 07:38 /tmp/myprog.HStsWoEi6373/
```

由于组与其他人都无法访问该目录，攻击者也无从得知你继续放入的文件名称，不过如果你的脚本是开放公众读取的，当然还是可能猜出来！由于目录无法列出成列表，所以没有权限的攻击者就无法确认他的猜测。

10.3.3 /dev/random与/dev/urandom特殊文件

有些系统会提供两种随机伪设备：/dev/random与/dev/urandom。现在这些仅在BSD系统、GNU/Linux、IBM AIX 5.2、Mac OS X与Sun Solaris 9，搭配两个第三方的实例与早期Solaris版本的计算机修整程序（注3）上，提供此支持。这些设备的任务，是提供永不为空的随机字节数据流：这样的数据来源是许多加密程序与安全应用程序所需要的。虽然已经有很多的简单算法可以产生这种虚拟随机数据流，但其实要产生一个真正

注3： 可到*http://www.cosy.sbg.ac.at/~andi/SUNrand/pkg/random-0.7a.tar.gz*取得，与参考*http://sunrpms.maraudingpirates.org/HowTo.html*。Sun提供了修补程序（10675[456]-01）到SUNWski包，让它们在旧式的Solaris里也能使用；在*http://sunsolve.sun.com/*可以找到它们。

的随机数据其实是很难的事：这部分请参考《Cryptographic Security Architecture: Design and Verification》（注 4）一书。

这两个设备的差别，在 /dev/random会一直封锁，直到系统所产生的随机数已充分够用，所以它可以确保高品质的随机数。相对地，/dev/urandom不会死锁，其数据的随机程度也不高（不过这已经足够通过大部分随机统计测试了）。

由于这些设备是共享资源，攻击者轻易就能加载拒绝服务，通过读取该设备并丢弃数据，阻断 /dev/random。现在比较一下这两个设备，请注意它们两个在 count 参数下的不同：

```
$ time dd count=1 ibs=1024 if=/dev/random > /dev/null     读取 1KB 的随机码元组
0+1 records in
0+1 records out
0.000u 0.020s 0:04.62 0.4%      0+0k 0+0io 86pf+0w

$ time dd count=1024 ibs=1024 if=/dev/urandom > /dev/null 读取 1MB 的随机码元组
1024+0 records in
2048+0 records out
0.000u 0.660s 0:00.66 100.0%    0+0k 0+0io 86pf+0w
```

/dev/random被读取的越多，它的响应越慢。我们用这两个设备在几个系统上实验，发现要自 /dev/random提取 10MB 的数据，竟耗掉一天或一天以上。而 /dev/urandom在我们最快的系统上执行，三秒钟即可产生相同的数据。

这两个伪设备都可以取代 mktemp，成为产生难以推测的临时性文件名的替代方案：

```
$ TMPFILE=/tmp/secret.$(cat /dev/urandom | od -x | tr -d ' ' | head -n 1)

$ echo $TMPFILE                              显示随机文件名
/tmp/secret.00000003024d462705664c043c04410e570492e
```

此处，我们从 /dev/urandom读取二进制字节数据流，以 od 将其转换为十六进制，使用 tr 去掉空格，之后在满一行时停止。因为 od 将每个输出行转换为 16 个字节，因而提供了 16 × 8 = 128 个随机位，作为副文件名，或是 2^{128}（约 3.40 × 10^{38}）种可能的副文件名。如果该文件名建立在仅用户可列出的目录中，则攻击者是无从猜测的。

10.4 寻找文件

Shell模式匹配的功能还不足以做到匹配递归整个文件树状结构里的文件，而ls与stat也没有提供Shell模式以外其他选定文件的方式。幸好，UNIX 还有其他工具，提供比这些命令更好的功能。

注 4： Peter Gutmann, Springer-Verlag, 2004, ISBN 0-387-95387-6。

10.4.1 快速寻找文件

locate首度问世是出现在Berkeley UNIX，到了GNU *findutils*包里（注5）又再重新实现。locate将文件系统里的所有文件名压缩成数据库，以迅速找到匹配类Shell通配字符模式的文件名，不必实际查找整个庞大的目录结构。这个数据库，通常是在半夜通过cron，在具有权限的工作中执行updatedb建立。locate对用户来说有其必要性，它可以回答用户：系统管理者究竟将gcc包放在何处?

```
$ locate gcc-3.3.tar                    寻找 gcc-3.3 版本
/home/gnu/src/gcc/gcc-3.3.tar-lst
/home/gnu/src/gcc/gcc-3.3.tar.gz
```

缺乏通配字符模式时，locate会报告含有将参数作为子字符串的文件；这里找到两个匹配的文件。

由于locate的输出量可能极多，它通常会通过管道丢给分页程序（pager），如less；或是查找过滤程序，例如grep处理：

```
$ locate gcc-3.3 | fgrep .tar.gz       寻找gcc-3.3，不过仅报告供流通用的存档文件
/home/gnu/src/gcc/gcc-3.3.tar.gz
```

通配字符模式须被保护，以避免Shell展开，这么一来locate才能自己处理它们：

```
$ locate '*gcc-3.3*.tar*'              在locate里，使用通配字符匹配，以寻找gcc-3.3
...
/home/gnu/src/gcc/gcc-3.3.tar.gz
/home/gnu/src/gcc/gcc-3.3.1.tar.gz
/home/gnu/src/gcc/gcc-3.3.2.tar.gz
/home/gnu/src/gcc/gcc-3.3.3.tar.gz
...
```

注意： locate或许不适用于所有站点，因为它会将被限制访问的目录下的文件名泄露给用户。如果有这点考虑，只需简单将updatedb的操作交给一般用户权限执行：这么一来，不合法的用户便无从得知原本就不该让它找到的文件名了。不过比较好的方式是使用*secure locate*包：slocate（注6），它也会将文件的保护与所有权存储在数据库里，但只显示用户可以访问的文件名。

---

updatedb提供选项，可建立文件系统里选定位置的locate数据库，例如用户的根目录树状结构，所以locate可用作个人文件的查询。

注5： 取自 *ftp://ftp.gnu.org/gnu/findutils/*。

注6： 取自 *ftp://ftp.geekreview.org/slocate/*。

10.4.2 寻找命令存储位置

偶尔你也可能会想知道，调用一个没有路径的命令时，它在文件系统的位置如何。Bourne-Shell 家族里的 type 命令可以告诉你：

```
$ type gcc                                   gcc 在哪?
gcc is /usr/local/bin/gcc

$ type type                                  type 是什么?
type is a Shell builtin

$ type newgcc                                newgcc 是什么?
newgcc is an alias for /usr/local/test/bin/gcc

$ type mypwd                                 mypwd 是什么?
mypwd is a function

$ type foobar                                这个（不存在的）命令是什么?
foobar not found
```

请注意，type 为内部 Shell 命令，所以它认得别名与函数。

我们在例8-1里展示过的pathfind命令，提供的是另一种查找整个目录路径下的方式，而不只是 type 查找的 PATH 列表。

10.4.3 find 命令

假定你想选择大于某个大小的文件，或是三天前修改过、属于你的文件，或者拥有三个或三个以上直接链接的文件，就会需要 UNIX 工具集里最强力的 find 命令。

find实例提供了60种之多的不同选项，所以我们讨论的只是其中一小部分而已。本段 find 板块概括的是几个比较重要的 find 选项。

如果你需要在整个目录树状结构分支里绕来绕去寻找某个东西，find 可以帮你完成此工作，不过你首先得好好地把整个使用手册读一遍，了解该怎么找。GNU版本的使用手册极其丰富，我们建议你深入研究。

10.4.3.1 使用 find 命令

find 与其他 UNIX 命令最大的不同处在于：要查找的文件与目录，要放在参数列表的第一位，且目录几乎是递归地向下深入（寻找）。最终要显示而选定名称的选项或操作放在命令行的最后。

find

语法

```
find [ files-or-directories ] [ options ]
```

用途

寻找与指定名称模式匹配于或具有给定属性的文件。

主要选项

注意内文介绍有关部分选项需接上数字 *mask* 与 *n* 的介绍：

`-atime` *n*

选定 *n* 天前访问的文件。

`-ctime` *n*

选定 *n* 天前改过 inode 的文件。

`-follow`

接着符号性连接。

`-group` *g*

选定组 *g* 内的文件（*g* 为用户组 ID 名称或数字）

`-links` *n*

选定拥有 *n* 个直接链接的文件。

`-ls`

产生类似 `ls` 冗长形式的列表，而不是只有文件名。

`-mtime` *n*

选定 *n* 天前修改过的文件。

`-name'`*pattern*`'`

选定文件名与Shell通配字符模式匹配的文件（通配字符模式会使用括号框起来，可避免Shell解释）。

`-perm` *mask*

选定与指定八进制权限掩码匹配的文件。

`-prune`

不向下递归到目录树状结构里。

`-size` *n*

选择大小为 *n* 的文件。

find（续）

-type *t*

选定类型*t*的文件，类型是单一字母：d为目录、f为文件、l为符号性连接。还有其他字母表示其他的文件类型，不过很少用到。

-user u

选定用户*u*拥有的文件（*u*为用户ID名称或编号）。

行为

find向下深入目录树状结构，寻找所有在这些目录树下的文件。接下来，应用其命令行选项所定义的选定器，选择文件供进一步操作，通常是显示它们的名称，或产生类似ls的冗长列出。

警告

由于find默认会向下寻找目录，所以当它在大型文件系统中寻找时，会花费很长的执行时间。

find输出的是未排序的结果。

find提供额外选项，可对选定的文件执行任意操作。由于这是一个潜在的危险，我们不建议使用这类选项，除非你的系统拥有严格的控制。

find不同于ls与Shell的地方是：它没有隐藏文件的概念，也就是说：就算是点号开头的文件名，find还是能找到它。

另一点不同于ls的是：find不排序文件名。它只是以它读到目录的顺序依次显示，事实上这个排序应是随机的(注7)。因此，你可能得在find命令之后，通过管道加入sort步骤。

最后一个与ls不同的地方是：当find处理的是目录时，它会自动递归深入目录结构，寻找在那之下的任何东西，除非你使用-prune选项要求不要这么做。

当find找到文件要处理时，它会先执行命令行选项所设置的选择限制，如果这些测试成功，则将名称交给内部的操作程序处理。默认操作是将名称打印在标准输出上，不过如果使用-exec选项，则可提供命令模板，在其中名称可以被替换，并再执行该命令。旧的find实现会要求明确地指出-print选项，才能产生输出，不过幸好这样的不良设计，已在现行所有实现中完成修正，至少我们测试过的都已完成修正，包括POSIX。

注7：因为用户习惯在ls与Shell通配字符展开下，看到排序后的列表，因此常常认定目录必以排序后的顺序存储名称。但如果你编写一个调用opendir()、readdir()以及closedir()程序库的程序，你就会发现qsort()也需要是可移植的！

在选定的文件上自动执行命令是很强的功能，但也极度危险。如果该命令具破坏性，那么最好是让 find 先将列表产生在临时性文件中，再由可胜任的人小心地确认，决定是否将命令进一步自动化处理。

使用 find 进行破坏性目的的 Shell 脚本，在编写时必须格外小心，之后，也必须彻底执行调试，例如在破坏性命令开始前插入 echo，这么一来你可以看看会有哪些操作，而不必真地执行它。

我们先来做一个最简单的例子：单纯使用 find 寻找当前目录树下的所有东西。正如前面的例子，我们先从空的目录开始，之后再将它填入一些空文件：

```
$ ls                                     确认这是一个空目录
$ mkdir -p sub/sub1                      建立一个目录树
$ touch one two .uno .dos                在该目录最上层建立一些空文件
$ touch sub/three sub/sub1/four          在树状结构较深层的地方建立一些空文件
$ find                                   从此开始寻找所有东西
.
./sub
./sub/sub1
./sub/sub1/four
./sub/three
./one
./two
./.uno
./.dos
```

这个混乱的列表可以很轻松地完成排序：

```
$ find | LC_ALL=C sort                    以传统顺序，排序 find 的输出结果
.
./.dos
./.uno
./one
./sub
./sub/sub1
./sub/sub1/four
./sub/three
./two
```

设置 LC_ALL 取得传统的（ASCII）排序顺序，这是因为现行 sort 实现都与 locale 相关，见 4.1.1 节。

find 还有一个好用的选项：-ls，可得到如指定了 ls -liRs 的输出结果。不过，它缺乏进一步的选项控制这个冗长显示的格式：

```
$ find -ls                                   寻找文件，并使用 ls 风格的输出结果
1451550   4 drwxr-xr--   3 jones   devel    4096  Sep 26 09:40 .
1663219   4 drwxrwxr-x   3 jones   devel    4096  Sep 26 09:40 ./sub
1663220   4 drwxrwxr-x   2 jones   devel    4096  Sep 26 09:40 ./sub/sub1
1663222   0 -rw-rw-r--   1 jones   devel       0  Sep 26 09:40 ./sub/sub1/four
1663221   0 -rw-rw-r--   1 jones   devel       0  Sep 26 09:40 ./sub/three
1451546   0 -rw-rw-r--   1 jones   devel       0  Sep 26 09:40 ./one
1451547   0 -rw-rw-r--   1 jones   devel       0  Sep 26 09:40 ./two
1451548   0 -rw-rw-r--   1 jones   devel       0  Sep 26 09:40 ./.uno
1451549   0 -rw-rw-r--   1 jones    devel      0  Sep 26 09:40 ./.dos

$ find -ls | sort -k11                                寻找文件，并以文件名排序
1451550   4 drwxr-xr--   3 jones   devel    4096  Sep 26 09:40 .
1451549   0 -rw-rw-r--   1 jones   devel       0  Sep 26 09:40 ./.dos
1451548   0 -rw-rw-r--   1 jones   devel       0  Sep 26 09:40 ./.uno
1451546   0 -rw-rw-r--   1 jones   devel       0  Sep 26 09:40 ./one
1663219   4 drwxrwxr-x   3 jones   devel    4096 Sep 26 09:40 ./sub
1663220   4 drwxrwxr-x   2 jones   devel    4096  Sep 26 09:40 ./sub/sub1
1663222   0 -rw-rw-r--   1 jones   devel       0  Sep 26 09:40 ./sub/sub1/four
1663221   0 -rw-rw-r--   1 jones   devel       0  Sep 26 09:40 ./sub/three
1451547   0 -rw-rw-r--   1 jones   devel       0  Sep 26 09:40 ./two
```

为作对照，这里我们以 ls 显示相同的文件 meta 数据：

```
$ ls -liRs *                                 显示 ls 递归的冗长输出
752964    0 -rw-rw-r--    1 jones    devel        0 2003-09-26 09:40 one
752965    0 -rw-rw-r--    1 jones    devel        0 2003-09-26 09:40 two

sub:
total 4
752963    4 drwxrwxr-x    2 jones    devel     4096 2003-09-26 09:40 sub1
752968    0 -rw-rw-r--    1 jones    devel        0 2003-09-26 09:40 three

sub/sub1:
total 0
752969    0 -rw-rw-r--    1 jones    devel        0 2003-09-26 09:40 four
```

现在我们给 find 命令一些文件模式：

```
$ find 'o*'                        寻找此目录下，以"o"开头的文件
one

$ find sub                         在目录 sub 下寻找文件
sub
sub/sub1
sub/sub1/four
sub/three
```

接下来，我们抑制目录向下寻找的功能：

```
$ find -prune                     不要在此目录下寻找
.

$ find . -prune                   相同操作的另一种方式
.
```

```
$ find * -prune                     寻找此目录下的文件
one
sub
two

$ ls -d *                           列出文件，但没有目录内容
one  sub  two
```

请注意：没有文件或目录参数，是等同于当前的目录，所以前两个例子只会报告该目录。然而，星号会匹配每一个非隐藏文件，所有第三个find的运行，就如同ls -d，只不过它是一行显示一个文件。

这时便是试试 find 几个强大选项的时候了。我们从所有者与组的选定开始：选项-group与-user都需要一个接着的符号名称或数值识别码。因此，find / -user root会启动执行很久的查找文件操作，它会在 root 拥有的整个树状结构中查找文件。除非此命令是由 root 执行，否则目录权限几乎一定会隐藏此树状结构的主要部分。

你可能会预期在登录目录树中的所有文件都属于你。要确认这件事，只要执行 find $HOME/. ! -user $USER 即可。惊叹号是非的意思，也就是说，这条命令就是：从我的根目录开始，列出所有不属于我的文件。HOME 与 USER 两个都是标准 Shell 变量，用于定制你的登录，所以这个命令适用于所有人。我们用 $HOME/. 而非只是 $HOME，使得如果 $HOME 为符号连接，命令也可正常运行。

-perm 需要接上一个八进制字符串的权限掩码，其可以具有选用的正/负号。如掩码不带任何正负号，则必须有确实的匹配权限。如果为负号，则所有的位设置都必须匹配。如果为正号，则至少须有一个位设置要匹配。看来有点复杂，我们将惯用的方式放在表10-1中。

表10-1：find的常见权限设置

选项	意义
-perm -002	寻找（所有者与组外的）其他人可写入的文件。
-perm -444	寻找任何人都可读取的文件。
! -perm -444	寻找任何人都无法读取的文件。
-perm 444	寻找权限为 r--r--r-- 的文件。
-perm +007	寻找其他人可访问的文件。
! -perm +007	寻找其他人无法访问的文件。

-size选项必须接上一个数字参数。默认值是以512字节为单位的大小，不过很多find实例，允许在数字之后加上c表示字符（字节），或k表示kilobyte（KB）。如果数字未带有正负号，则指的是必须确实匹配于该文件大小。如果为负，则只有小于该数字（绝

对值）的文件大小是匹配的。否则，带有正号，则只有大于该容量的文件才匹配。所以，find $HOME/. -size +1024k会在你登录目录树下的所有文件中，寻找是否有大于1MB的，而find . -size 0则是寻找当前目录下的所有文件中是空的。

-type选项必须接上一个单词母参数，以标明文件类型。较重要的几个为d的目录、f的一般文件，以及l的符号连接。

-follow选项要求find接上符号连接，你可以用此来找出断掉的连接：

```
$ ls                                显示我们有一个空目录下

$ ln -s one two                     为不存在的文件建立软性（符号性）连接

$ file two                          诊断此文件
two: broken symbolic link to one

$ find .                            寻找所有文件
.
./two

$ find . -type l                    只找软性连接
./two

$ find . -type l -follow            寻找软连接，并试图跟随它们
find: cannot follow symbolic link ./two: No such file or directory
```

-links选项要求接上一个整数。如未指定正负号，它会只选择具有指定数量的直接连接的文件；如果为负号，则只寻找连接数小于该数字（绝对值）的文件；如果为正号，则仅选择连接数大于该数的文件。因此，如果你要寻找具有直接链接的文件，通常是这样：find . -links +1。

-atime（访问时间）、-ctime（inode变更时间）与-mtime（修改时间）选项必须接上一个以天为单位的整数。未指定正负号，即指确实的几天前；如果为负，则指少于该天数（绝对值）；为正，则为大于该天数。一般惯用法为find . -mtime -7可寻找一周前修改过的文件。

警告： 可惜的是，find不允许数目是部分（分数）的，或是单位字尾的：我们常会需要以年、月、周、时、分或秒为这些选项的单位。GNU find提供的-amin、-cmin，与-mmin，可以分钟为单位，但是在原始时间戳选择选项上的单位字尾应该是要更一般化的。

有个相关的选项-newer *filename*，可以仅选择比指定文件更接近最近时间修改过的文件。如果你要的单位比这个时间还精细，可以建立一个空文件touch -t *date_time*

timestampfile，然后以此文件搭配使用 -newer 选项。如果你要找的是比该文件更旧的文件，使用否定选项即可：! -newer *timestampfile*。

find 命令的选择器选项也可合并使用：也就是所有的匹配都匹配，才采取操作。你也可以另外配置 -a（AND）选项，而 -o（OR）选项也可用于标明在其左右两边的匹配中至少有一组匹配的情况。下面便是应用这些布尔运算符的两个例子，：

```
$ find . -size +0 -a -size -10        寻找文件大小块小于 10 (5120 字节) 的非空文件
...

$ find . -size 0 -o -atime +365       寻找空文件，或过去一年都未读取过的文件
...
```

-a 与 -o 运算符，配上组选项 \ (与 \ )，可用以建立更复杂的布尔选择器。你应该很少需要用到，而当你使用时，会发现它们已复杂到：一旦它们需要被调试时，你得在脚本中隐藏它们，然后在调试完后再使用该脚本。

10.4.3.2 find的简易版脚本

到目前为止，我们已使用 find 产生匹配特定选择需求的文件列表，还可以设法将它们送进一个简单的管道。现在，让我们来看看更复杂一点的例子。在3.2.7.1节里，我们介绍过简单的 sed 脚本，可将 HTML 转换为 XHTML：

```
$ cat $HOME/html2xhtml.sed               显示将 HTML 转换为 XHTML 的 sed 命令
s/<H1>/<h1>/g
s/<H2>/<h2>/g
...
s:</H1>:</h1>:g
s:</H2>:</h2>:g
...
s:</[Hh][Tt][Mm][LL]>:</html>g
s:</[Hh][Tt][Mm][Ll]>:</html>:g
s:<[Bb][Rr]>:<br/>:g
...
```

这样的脚本可以将 HTML 转换为 XHTML（HTML 的标准化以 XML 为主的版本）的绝大部分工作自动化。将 sed 结合 find，辅以简单的循环可以让工作减少为只有下面几行代码：

```
cd top level web site directory
find . -name '*.html' -type f |                          寻找所有 HTML 文件
    while read file                                      将文件名读进变量里
    do
        echo $file                                       打印处理进度
        mv $file $file.save                              存储备份副本
        sed -f $HOME/html2xhtml.sed < $file.save > $file 开始变更
    done
```

10.4.3.3 find 的复杂版脚本

本节，我们将更纯熟地应用 find，开发一个真正实用的范例（注 8）。这个 Shell 脚本叫作 filesdirectories，它是针对具有大型根目录树状结构的部分本地用户，在夜间通过 crontab 系统（见 13.6.4 节），整理之前修改过的文件，以天数划分成组，建立文件与目录的多个列表。这样做有助于提醒他们近期做过的事，且提供的是一个更快的方式，也就是他们只需要查找单一的列表文件，就能在他们的目录结构下找到特定文件，而不必确实地寻找整个文件系统本身。

filesdirectories 必须使用 GNU 的 find 以便使用 -fprint 选项，该选项允许在一次通过整个目录树下建立多个输出文件，这样的脚本可以较原始 UNIX find 的多重调用版本，高出 10 倍速（*tenfold speedup*）。

这段脚本将由一般安全性功能开始：在 #! 行内标明 - 选项，见 2.4 节：

```
#! /bin/sh -
```

设置 IFS 变量为换行符号（空格）制表字符（newline-space-tab）：

```
IFS='
        '
```

并设置 PATH 变量，以确保先找 GNU 的 Find：

```
PATH=/usr/local/bin:/bin:/usr/bin    # 需要GNU find的-fprint选项
export PATH
```

接着，确认参数是否为预期的单一参数，否则，显示简短的错误信息到标准错误输出，并以非零状态值离开：

```
if [ $# -ne 1 ]
then
        echo "Usage: $0 directory" >&2
        exit 1
fi
```

作为最后一项安全性功能，这段脚本引用 umask 限制仅输出文件的所有者可以访问：

```
umask 077                             # 确保文件私密性
```

filesdirectories 允许 TMPDIR 环境变量覆盖默认的临时性文件目录：

```
TMP=${TMPDIR:-/tmp}                   # 允许另一个临时性目录
```

下一步是将 TMPFILES 初始化为一长串收集输出的临时性文件列表：

注 8： 感谢 University，of Utah 的 Pieter J. Bowman 贡献。

```
TMPFILES="
        $TMP/DIRECTORIES.all.$$ $TMP/DIRECTORIES.all.$$.tmp
        $TMP/DIRECTORIES.last01.$$ $TMP/DIRECTORIES.last01.$$.tmp
        $TMP/DIRECTORIES.last02.$$ $TMP/DIRECTORIES.last02.$$.tmp
        $TMP/DIRECTORIES.last07.$$ $TMP/DIRECTORIES.last07.$$.tmp
        $TMP/DIRECTORIES.last14.$$ $TMP/DIRECTORIES.last14.$$.tmp
        $TMP/DIRECTORIES.last31.$$ $TMP/DIRECTORIES.last31.$$.tmp
        $TMP/FILES.all.$$ $TMP/FILES.all.$$.tmp
        $TMP/FILES.last01.$$ $TMP/FILES.last01.$$.tmp
        $TMP/FILES.last02.$$ $TMP/FILES.last02.$$.tmp
        $TMP/FILES.last07.$$ $TMP/FILES.last07.$$.tmp
        $TMP/FILES.last14.$$ $TMP/FILES.last14.$$.tmp
        $TMP/FILES.last31.$$ $TMP/FILES.last31.$$.tmp
        "
```

这些输出文件包括了整个树状结构下（*.all.*）的目录与文件之名称，以及在前一天（*.last01.*）、前两天（*.last02.*）等那些修改过的名称。

WD 变量存储参数目录名称，供稍后使用，然后脚本会变更到该目录：

```
WD=$1
cd $WD || exit 1
```

在执行 find 之前变更工作中目录，可解决两个问题：

- 如果参数非目录，或是但缺乏必需的权限，那么 cd 命令会失败，脚本会立即以非零离开值而终止。
- 如果参数为符号连接，则 cd 会按照这个连接找到真正的位置。find 如果未给定额外选项，是不会跟随符号连接的，但是没有方法可以告诉它只为顶层目录这么做。实际上，我们不要 filesdirectories 按照目录树里的连接，尽管增加一个选项以如此做并不难。

trap 命令确保临时性文件会在脚本终止时被删除：

```
trap 'exit 1'          HUP INT PIPE QUIT TERM
trap 'rm -f $TMPFILES' EXIT
```

离开状态值会在跨过 EXIT 捕捉之后仍被保留，见 13.3.2。

接下来的部分就是最精彩，也是最困难的工作了，多行 find 命令。使用 -name 选项的行，会匹配来自前次执行的输出文件名称，而 -true 选项会忽略这些操作，以免这些信息弄乱输出报告：

```
find . \
           -name DIRECTORIES.all -true \
        -o -name 'DIRECTORIES.last[0-9][0-9]' -true \
        -o -name FILES.all -true \
        -o -name 'FILES.last[0-9][0-9]' -true \
```

下一行会匹配所有的一般文件，并利用 -fprint 选项，将它们的名称写到 $TMP/FILES.all.$$：

```
        -o -type f          -fprint $TMP/FILES.all.$$ \
```

接下来的5行，分别选定31、14、7、2以及1天前修改过的文件（-type f 选择器仍有效），再以 -fprint 选项将它们的名称写到指定的临时性文件：

```
        -a        -mtime -31 -fprint $TMP/FILES.last31.$$ \
        -a        -mtime -14 -fprint $TMP/FILES.last14.$$ \
        -a        -mtime  -7 -fprint $TMP/FILES.last07.$$ \
        -a        -mtime  -2 -fprint $TMP/FILES.last02.$$ \
        -a        -mtime  -1 -fprint $TMP/FILES.last01.$$ \
```

测试操作是由最旧到最新依次完成，因为每一组文件，都是前一组的子集合，在每一步骤逐步减少处理量。因此10天前的文件会通过前两个 -mtime 测试，但是会使得接下来的三个失败，所以，它只会被包括在FILES.last31.$$与FILES.last14.$$文件里。

下一行乃匹配目录，并使用-fprint选项将它们的名称写到$TMP/DIRECTORIES.all.$$：

```
        -o -type d          -fprint $TMP/DIRECTORIES.all.$$ \
```

find 命令的最后5行匹配目录的子集合（仍使用 -type d 选择器），再将它们的名称写到输出文件：

```
        -a        -mtime -31 -fprint $TMP/DIRECTORIES.last31.$$ \
        -a        -mtime -14 -fprint $TMP/DIRECTORIES.last14.$$ \
        -a        -mtime  -7 -fprint $TMP/DIRECTORIES.last07.$$ \
        -a        -mtime  -2 -fprint $TMP/DIRECTORIES.last02.$$ \
        -a        -mtime  -1 -fprint $TMP/DIRECTORIES.last01.$$ \
```

当 find 命令结束时，它的初步报告可在临时性文件中获得，只不过还没存储。然后脚本会循环处理这些报告文件，最后结束工作：

```
    for i in FILES.all FILES.last31 FILES.last14 FILES.last07 \
            FILES.last02 FILES.last01 DIRECTORIES.all \
            DIRECTORIES.last31 DIRECTORIES.last14 \
            DIRECTORIES.last07 DIRECTORIES.last02 DIRECTORIES.last01
    do
```

sed 会将每个报告行前置的 ./ 替换为用户指定的目录名称，所以输出文件会包含完整 —— 而非相对的路径：

```
        sed -e "s=^[.]/=$WD/=" -e "s=^[.]$=$WD=" $TMP/$i.$$ |
```

sort将sed的结果进行排序，传入临时性文件，并将文件命名为：输入文件名加上 .tmp 结尾：

```
LC_ALL=C sort > $TMP/$i.$$.tmp
```

将LC_ALL设为C可产生传统UNIX排序顺序，也就是我们长期以来惯于使用的，这么做可避免设置为较现代的locale时可能造成的混淆与非预期情况。因为我们的系统各有不同的默认locale，所以使用传统顺序，在我们互异的环境下，特别有帮助。

cmp命令为默认检查报告文件是否不同于前次执行的报告文件，如果不同，则换掉旧的：

```
cmp -s $TMP/$i.$$.tmp $i || mv $TMP/$i.$$.tmp $i
```

否则，留下临时性文件，由trap处理器进行清除操作。

脚本的最后一个语句，会完成处理报告文件的循环：

```
done
```

执行期间，脚本通过之前设置的EXIT捕捉而终止。

完整的filesdirectories脚本见例10-1。其架构清晰，足以让你轻松地略作修改，即可产生其他报告文件，例如15分钟前、半年前或一年前修改过的文件和目录。改变-mtime值的正负号，即可取得最近没有修改过的文件的报告，这对于找出过时的文件很有用。

例10-1：find的复杂版Shell脚本

```
#! /bin/sh -
# 寻找所有的文件及目录，
# 在目录树下，将最近修改过的加以组化，
# 并于最上层的 FILES.* 与 DIRECTORIES.* 内置立列表。
#
# 语法 :
#     filesdirectories directory

IFS='
        '

PATH=/usr/local/bin:/bin:/usr/bin    # 需要GNU find的-fprint选项
export PATH

if [ $# -ne 1 ]
then
        echo "Usage: $0 directory" >&2
        exit 1
fi

umask 077                            # 确保文件隐私
TMP=${TMPDIR:-/tmp}                  # 允许另一个临时性目录
TMPFILES="
        $TMP/DIRECTORIES.all.$$ $TMP/DIRECTORIES.all.$$.tmp
        $TMP/DIRECTORIES.last01.$$ $TMP/DIRECTORIES.last01.$$.tmp
```

```
        $TMP/DIRECTORIES.last02.$$ $TMP/DIRECTORIES.last02.$$.tmp
        $TMP/DIRECTORIES.last07.$$ $TMP/DIRECTORIES.last07.$$.tmp
        $TMP/DIRECTORIES.last14.$$ $TMP/DIRECTORIES.last14.$$.tmp
        $TMP/DIRECTORIES.last31.$$ $TMP/DIRECTORIES.last31.$$.tmp
        $TMP/FILES.all.$$ $TMP/FILES.all.$$.tmp
        $TMP/FILES.last01.$$ $TMP/FILES.last01.$$.tmp
        $TMP/FILES.last02.$$ $TMP/FILES.last02.$$.tmp
        $TMP/FILES.last07.$$ $TMP/FILES.last07.$$.tmp
        $TMP/FILES.last14.$$ $TMP/FILES.last14.$$.tmp
        $TMP/FILES.last31.$$ $TMP/FILES.last31.$$.tmp
        "

WD=$1
cd $WD || exit 1

trap 'exit 1'           HUP INT PIPE QUIT TERM
trap 'rm -f $TMPFILES' EXIT

find . \
           -name DIRECTORIES.all -true \
        -o -name 'DIRECTORIES.last[0-9][0-9]' -true \
        -o -name FILES.all -true \
        -o -name 'FILES.last[0-9][0-9]' -true \
        -o -type f              -fprint $TMP/FILES.all.$$ \
        -a         -mtime -31 -fprint $TMP/FILES.last31.$$ \
        -a         -mtime -14 -fprint $TMP/FILES.last14.$$ \
        -a         -mtime  -7 -fprint $TMP/FILES.last07.$$ \
        -a         -mtime  -2 -fprint $TMP/FILES.last02.$$ \
        -a         -mtime  -1 -fprint $TMP/FILES.last01.$$ \
        -o -type d              -fprint $TMP/DIRECTORIES.all.$$ \
        -a         -mtime -31 -fprint $TMP/DIRECTORIES.last31.$$ \
        -a         -mtime -14 -fprint $TMP/DIRECTORIES.last14.$$ \
        -a         -mtime  -7 -fprint $TMP/DIRECTORIES.last07.$$ \
        -a         -mtime  -2 -fprint $TMP/DIRECTORIES.last02.$$ \
        -a         -mtime  -1 -fprint $TMP/DIRECTORIES.last01.$$

for i in FILES.all FILES.last31 FILES.last14 FILES.last07 \
        FILES.last02 FILES.last01 DIRECTORIES.all \
        DIRECTORIES.last31 DIRECTORIES.last14 \
        DIRECTORIES.last07 DIRECTORIES.last02 DIRECTORIES.last01
do
        sed -e "s=^[.]/=$WD/=" -e "s=^[.]$=$WD=" $TMP/$i.$$ |
                LC_ALL=C sort > $TMP/$i.$$.tmp
        cmp -s $TMP/$i.$$.tmp $i || mv $TMP/$i.$$.tmp $i
done
```

10.4.4 寻找问题文件

在10.1节里，我们注意到包含特殊字符（如换行字符）的文件名有点麻烦。GNU `find` 具有`-print0`选项，以显示文件名为NUL终结的字符串。由于路径名称可包含任何字符，除了NUL以外，所以这个选项，可产生能够被清楚解析的文件名列表。

使用典型的UNIX工具很难去剖析这种列表，因为它们大部分都假定是行导向的文字输入。然而，在使用一次一个字节（byte-at-a-time）输入的编译语言中，例如C、C++或Java，会很直觉地编写一个程序诊断文件系统里是否有问题的文件名。有时它们只是单纯程序员的错误，有时是攻击者通过伪装文件名隐藏他们的存在而放在那里的。

假设你执行目录列出，且得到这样的结果：

```
$ ls                                    列出目录
 .   ..
```

第一眼，你可能觉得没问题啊，因为我们知道空目录总是会包含两个特殊的隐藏点号文件，指的是当前目录与父目录。然而，请注意这里我们并未使用 -a 选项，所以我们不应该看到任何隐藏文件，而且在输出的第一个点号之前出现一个空格，有点不对劲！让我们用 find 和 od 作进一步调查吧：

```
$ find -print0 | od -ab                将以 NUL 终结的文件名，转换为八进制及 ASCII
0000000   . nul   .   /  sp   . nul   .   /  sp   .   . nul   .   /   .
        056 000 056 057 040 056 000 056 057 040 056 056 000 056 057 056
0000020  nl nul   .   /   .   .  sp   .   .  sp   .   .  sp   .  sp  nl
        012 000 056 057 056 056 040 056 056 040 056 056 040 056 040 012
0000040  nl  nl  sp  sp nul
        012 012 040 040 000
0000045
```

我们通过tr的帮助，让这个信息更具可读性，将空格转换为S、换行字符转为N，而NUL则变成换行符号：

```
$ find -print0 | tr ' \n\0' 'SN\n'     将问题字符转换为可见的S与N
.
./S.
./S..
./.N
./..S..S..S.SNNNSS
```

现在知道发生什么事了：我们有一个一般的点号目录，然后有一个文件叫作“空格——点号”，另一个则是“空格——点号——点号”，另有叫作“点号——换行符号”，以及最后一个叫作“点号——点号——空格——点号——点号——空格——点号——点号——空格——点号——空格——换行符号——换行符号——换行符号——空格——空格”的文件。除非有人正在你的文件系统里练习摩斯码（Morse code），否则这些文件看起来令人可疑，因此在去除这些文件之前，你应该先做一番调查。

10.5 执行命令：xargs

当find产生一个文件列表时，将该列表提供给另一个命令有时是很有用的。通常，这

是通过Shell的命令替换功能完成，就像下面：在系统标头文件里，查找POSIX_OPEN_MAX符号：

```
$ grep POSIX_OPEN_MAX /dev/null $(find /usr/include -type f | sort)
/usr/include/limits.h:#define    _POSIX_OPEN_MAX           16
```

当你编写程序或使用命令，处理这样一串对象列表时，如果列表为空，你也应该确保它的行为是适当的。因为grep在没有给定任何文件参数的情况下，会读取标准输入，所以我们可以提供/dev/null这样的参数，以确保不会因为find未产生输出，而等待终端输入悬在那。在这里不会发生这种情况，但开发防御性程序的习惯是好的。

来自替换命令所产生的输出有时会很长，可能还会遇到在kernel里，因为对命令行的结合长度的限制，以及超出其环境变量的问题。发生这种情况时，你会看到：

```
$ grep POSIX_OPEN_MAX /dev/null $(find /usr/include -type f | sort)
/usr/local/bin/grep: Argument list too long.
```

你可以通过getconf查询该限制：

```
$ getconf ARG_MAX                          取得 ARG_MAX 的系统组态值
131072
```

在我们测试过的系统中，报告值的范围从最少24 576（IBM AIX）到最多1 048 320（Sun Solaris）。

ARG_MAX问题的解决方案就看xargs了：它可以在标准输入上取得参数列表、一行一个，再将它们以适当大小组起来（由主机的ARG_MAX值决定）传给另一个命令，此命令再作为xargs的参数。下面范例，即可剔除讨厌的Argument list too long的错误：

```
$ find /usr/include -type f | xargs grep POSIX_OPEN_MAX /dev/null
/usr/include/bits/posix1_lim.h:#define  _POSIX_OPEN_MAX        16
/usr/include/bits/posix1_lim.h:#define  _POSIX_FD_SETSIZE      _POSIX_OPEN_MAX
```

这里的/dev/null参数可确保grep总是会看到至少两个文件参数，使它于每个报告匹配的起始处，打印文件名。如果xargs未取得输入文件名，则它会默认地终止，甚至不会调用它的参数程序。

GNU的xargs支持--null选项：可处理GNU find的-print0选项所产生的NUL结尾的文件名列表。xargs将每个这样的文件名作为一个完整参数，传递给它执行的命令，而没有Shell（错误）解释问题或换行符号混淆的危险；然后是交给该命令处理它的参数。

xargs的选项可以控制哪个参数需要被替换，还可以限制传递给参数命令的一次引用所

使用的参数个数。GNU的版本甚至可以并行处理/执行多个参数进程。然而大多数时候，我们这里介绍的简单形式已足够应付了。如需要进一步的细节，或参考更复杂之功能与技巧的范例，可参考 *xargs*(1)的手册页。

10.6 文件系统的空间信息

辅以适当的选项，find 与 ls 命令可以报告文件大小，所以加上简短的 awk 程序协助，可以得到文件占据了多少字节的报告：

```
$ find -ls | awk '{Sum += $7} END {printf("Total: %.0f bytes\n", Sum)}'
Total: 23079017 bytes
```

然而，这样的报告低估了空间的使用，因为文件以固定大小的块（block）配置，它并未告诉我们整个文件系统里已用了多少及还可用多少空间。有两个好用的工具提供更完美的解决方案：df 与 du。

10.6.1 df 命令

df（disk free，磁盘可用空间）提供单行摘要，一行显示一个加载的文件系统的已使用的和可用的空间。其单位视系统而定，有些使用块，有些则是 kilobytes（KB）。大部分现代实现都支持 -k 选项，也就是强制使用 kilobyte 单位，以及 -l（小写 L）选项，仅显示本地文件系统，排除网络加载的文件系统。下面是我们自某个网页服务器输出的传统范例：

```
$ df -k
Filesystem           1K-blocks      Used Available Use% Mounted on
/dev/sda5              5036284   2135488   2644964  45% /
/dev/sda2                38890      8088     28794  22% /boot
/dev/sda3             10080520   6457072   3111380  68% /export
none                    513964         0    513964   0% /dev/shm
/dev/sda8               101089      4421     91449   5% /tmp
/dev/sda9             13432904    269600  12480948   3% /var
/dev/sda6              4032092   1683824   2143444  44% /ww
```

GNU 的 df 提供 -h（human-readable，人们易于理解的）选项，产生更简洁，但可能较令人混淆的报告：

```
$ df -h
Filesystem            Size  Used Avail Use% Mounted on
/dev/sda5             4.9G  2.1G  2.6G  45% /
/dev/sda2              38M  7.9M   29M  22% /boot
/dev/sda3             9.7G  6.2G  3.0G  68% /export
none                  502M     0  502M   0% /dev/shm
/dev/sda8              99M  4.4M   90M   5% /tmp
```

```
/dev/sda9              13G  264M   12G   3% /var
/dev/sda6             3.9G  1.7G  2.1G  44% /ww
```

输出行任意排列，不过因为有一行标头，又得保留它，所以要应用 sort 排序就变得较难。幸好，绝大多数系统上这样的输出都只有几行而已。

你还可以提供一个或多个文件系统名称或加载点的一份列表，来限制输出项目：

```
$ df -lk /dev/sda6 /var
Filesystem           1K-blocks      Used Available Use% Mounted on
/dev/sda6              4032092   1684660   2142608  45% /ww
/dev/sda9             13432904    269704  12480844   3% /var
```

df

语法

df [*options*] [*files-or-directories*]

用途

显示一个或多个文件系统内的 inode 或空间使用情况。

主要选项

-i

显示 inode 计数，而非空间。

-k

显示空间时，以 kilobyte（KB）为单位，而非块。

-l

小写 L，仅显示本地文件系统。

行为

df 会针对每个文件或目录参数，如果无提供参数，则为所有的文件系统，产生单行标头以识别输出栏，再接上包含该文件或目录的文件系统之使用量报告。

警告

每个系统上的 df 输出各异，因此如果使用在必须具可移植性的 Shell 脚本上会很不可靠。

df 的输出是未排序的。

针对远端文件系统所做的空间报告可能不尽然完全正确。

报告仅为当时（快照）的系统状态，对于运行中的多用户系统而言，在极短的时间内就可能会有完全不同的结果。

对于加载网络的文件系统，在`Filesystem`栏里的实例记录会前置主机名称（*hostname:*），这么一来，某些`df`实例，便会为了适应栏宽而切分为两行，对其他软件来说，要解析这样的输出信息相当棘手。下面就是Sun Solaris系统下的例子：

```
$ df
Filesystem   1k-blocks      Used Available Use% Mounted on
...
/dev/sdd1     17496684  15220472   1387420  92% /export/local
fs:/export/home/0075
              35197586  33528481   1317130  97% /a/fs/export/home/0075
...
```

`df`中关于远端文件系统的可用空间报告不尽完全正确，这是由于软件实例在计算供紧急情况使用所保留的空间不同所致。

在附录B中，我们会讨论文件系统中有关inode表格的议题，inode表格为固定不变大小，且在文件系统创建时即已设置。`-i`（inode unit，inode单位）选项提供访问inode使用量的一种方式。下面为同一台网页服务器下的范例：

```
$ df -i
Filesystem             Inodes   IUsed   IFree IUse% Mounted on
/dev/sda5              640000  106991  533009   17% /
/dev/sda2               10040      35   10005    1% /boot
/dev/sda3             1281696  229304 1052392   18% /export
none                   128491       1  128490    1% /dev/shm
/dev/sda8               26104     144   25960    1% /tmp
/dev/sda9             1706880     996 1705884    1% /var
/dev/sda6              513024  218937  294087   43% /ww
```

`/ww`文件系统是最完美的状态：因为它的inode使用与文件系统空间，两者都保留40%以上的容量可用。对一个健康的计算机系统而言，系统管理者应该例行地监控所有本地文件系统上inode的使用量。

`df`命令在选项与输出外表中有很大的差异，这对于想分析其输出结果的可移植程序来说，是相当麻烦的。Hewlett-Packard在HP-UX上的实现更是完全不同，不过幸好HP提供了与Berkeley风格相当的`bdf`，它会产生类似于我们范例的那种输出。要处理这种差异，我们建议在你的站点上的每一处都安装GNU版本；该命令为*coreutils*包的一部分，可参考4.1.5节的说明。

10.6.2 du命令

`df`会摘要文件系统的可用空间，但它并不会告诉你某个特定的目录树需要多少空间，这是`du`（disk usage；磁盘用量）的工作。`du`就像`df`一样：各系统间所使用的选项都不尽相同，且其空间单位可能也不一样。有两个常见的重要选项实现：`-k`（kilobyte单位）与`-s`（摘要）。这里是我们的网页服务器系统的例子：

```
$ du /tmp
12       /tmp/lost+found
1        /tmp/.font-UNIX
24       /tmp

$ du -s /tmp
24       /tmp

$ du -s /var/log /var/spool /var/tmp
204480   /var/log
236      /var/spool
8        /var/tmp
```

GNU 的版本提供 -h（human-readable，人类易于理解的）选项：

```
$ du -h -s /var/log /var/spool /var/tmp
200M     /var/log
236k     /var/spool
8.0k     /var/tmp
```

du不会对同一个文件计算额外的直接连接，且通常会忽略软性连接。然而，有些实例提供选项可强制跟随软性连接，不过选项名称各异：请参考你系统里的手册页。

du

语法

du [*options*] [*files-or-directories*]

用途

显示一个或多个目录树内的空间使用率。

主要选项

-k

空间的显示，以kilobyte（KB）为单位，而非（与系统相依的）块（block）。

-s

为每个参数，仅显示单行摘要。

行为

du 会针对每个文件或目录参数 —— 如果无提供这类参数则为当前目录，产生一个输出行，其会包含以整数表示的使用率，并接着文件或目录的名称。除非给定 -s 选项，否则每个目录参数会以递归方式被查找，为每个嵌套目录产生一个报告行。

警告

du 的输出未被排序。

du可以解决的一个常见问题是：找出是哪个用户用掉最多的系统空间。假定用户的根目录全放在 /home/users 下，root 可以这么做：

```
# du -s -k /home/users/* | sort -k1nr | less     找出大的根目录树
```

这么做可以产生使用空间前几名的列表，由最多到最少。在一些大型目录树下的 find *dirs* -size +10000 命令，可以迅速地找出可能是要压缩或删除的候选文件，且du的输出可以识别出最好搬到更大空间的用户目录树。

注意： 有些管理人员会将定期处理du报告并寄送警告邮件给使用过多目录树空间的用户这些工作加以自动化，像我们在第 7 章例 7-1 所编写的脚本那样。在我们的经验里，这么做会比使用文件系统配额（quota）功能还好（见 *quota*(1)使用手册）；因为它避免了指定特定数字（文件系统－空间的限制）给用户，那些限额的数字永远不会正确，且它们迟早会阻碍用户完成正常工作。

---

du的运行没有什么魔法，它就像其他程序那样，深入查找文件系统，再将每个文件的使用空间求和。因此在大型系统下执行，可能会有点慢，且通过严格的权限可锁住对目录树的查找；如果它的输出包含 Permission denied的信息，它的报告则无法充分计算空间使用率。通常，只有 root 有足够的权限，可以在本地系统的任何地方使用 du。

10.7 比较文件

本节，我们将会讨论比较文件领域里的四个相关主题：

- 检查两个文件是否相同，如果不同，找出哪里不同
- 应用两个文件的不同之处，使从其中一个回复另外一个
- 使用校验和（checksum）找出相同一致的文件
- 使用数字签名以验证文件

10.7.1 好用工具 cmp 与 diff

在文字处理上，最常出现的问题应该就是比较两个或两个以上的文件，看看它们的内容是否相同 —— 即便它们的名称不同。

如果你手上已经有两个要拿来比较的文件，那么文件比较的工具 cmp 马上能为你解答：

```
$ cp /bin/ls /tmp                    制作 /bin/ls 的私用副本
```

```
$ cmp /bin/ls /tmp/ls                    拿原始文件与副本比较
                                         没有任何输出，表示这两个文件一致
$ cmp /bin/cp /bin/ls                    比较两个不同的文件
/bin/cp /bin/ls differ: char 27, line 1  输出结果指出第一个不同处的位置
```

cmp 发现两个参数文件一致时，会采用默认的方式。如果你只对它的离开状态有兴趣，可以使用 -s 选项，抑制警告信息：

```
$ cmp -s /bin/cp /bin/ls                 默认地比较两文件的不同

$ echo $?                                显示离开码
1                                        非零，表示两个文件不同
```

如果你想知道两个相似的文件有何不同，可使用 diff：

```
$ echo Test 1 > test.1                   建立第一个 test 文件

$ echo Test 2 > test.2                   建立第二个 test 文件

$ diff test.[12]                         比较这两个文件
1c1
< Test 1
---
> Test 2
```

使用 diff 的惯例是：将旧文件作为第一个参数。

不同的行会以前置左角括号的方式，对应到左边的（第一个）文件，而前置右角括号则指的是右边的（第二个）文件。最前面的 1c1 为输入文件行编号的简洁表示方式，指出不同之处以及需要编辑的操作：在这里，c 表示改变（change）。在大一点的例子下，你还可能发现 a 是增加（add），与 d 是删除（delete）之意。

diff 的输出是仔细设计过的，因此其他程序可使用它的输出数据。例如版本修订控制系统（revision control system）就使用 diff 管理文件连续版本之间的差异。

有时，与 diff 系出同门的 diff3 也是相当好用的工具，它的任务与 diff 稍有不同：diff3 比较的是三个文件，例如基本版与由两个不同的人所做出来的两个修改文件，它还会产生一个 ed 命令的脚本，让用户将两组修改文件合并到基本版里。我们在这不多解释，有兴趣的读者可参考 *diff3*(1)手册页，以找到更多的范例。

10.7.2 patch 工具程序

patch 工具程序可利用 diff 的输出，结合原始文件，以重建另一个文件。因为相异的部分，通常比原始文件小很多，软件开发人员常会通过 email 交换相异处的列表，再使用 patch 应用它。下面的例子便是要告诉你，patch 如何将 test.1 的内容转换为那些匹配于 test.2 的内容：

```
$ diff -c test.[12] > test.dif           将相异处的相关内文，存储到 test.dif

$ patch < test.dif                       应用不同之处
patching file test.1

$ cat test.1                             显示修补后（patched）的 test.1 文件
Test 2
```

patch尽可能套用不同之处，然后报告失败的部分，由你自行手动处理。

虽然patch可使用diff的一般输出，但较通用的方式应是使用diff的-c选项，以取得上下文差异（context difference）处。这么做会产生较详细冗长的报告，让patch知道文件名，允许它验证变更位置，并回复不匹配之处。如果两个文件自从差异处已被记录下来之后都未有更改，则上下文差异功能是不重要的，但是在软件开发中，时常会有其中之一牵涉其中。

10.7.3 文件校验和匹配

要是你怀疑可能有许多文件具有相同的内文，而使用cmp或diff进行所有成对的比较，导致所花费的执行时间会随着文件数目增加成次方的增长，你马上就会受不了。

你可以使用file checksum（文件校验和），取得近似线性的性能。有很多工具可用来计算文件与字符串的校验和，包括sum、cksum，以及checksum（注9），消息摘要工具（注10）md5与md5sum，安全性散列（secure-hash）算法（注11）工具sha、sha1sum、sha256以及sha384。可惜的是：sum的实例在各平台间都不相同，使得它们的输出无法跨越不同的UNIX版本进行文件校验和的比较。cksum在OSF/1系统上的原始版本所产生的校验和不同于其他系统下的版本。

旧式的sum命令除外，这些程序里只有少数几个可以在系统之外找到，不过它们都很容易构建与安装。它们的输出格式互异，但传统上应是这样：

```
$ md5sum /bin/l?
696a4fa5a98b81b066422a39204ffea4  /bin/ln
cd6761364e3350d010c834ce11464779  /bin/lp
351f5eab0baa6eddae391f84d0a6c192  /bin/ls
```

注9：可到 *http://www.math.utah.edu/pub/checksum/*。

注10：R. Rivest, RFC 1321:《The MD5 Message-Digest Algorithm》，可参考 *ftp://ftp.internic.net/rfc/ rfc1321.txt*。md5sum是 GNU *coreutils* 包的一部分。

注11：NIST, FIPS PUB 180-1：《Secure Hash Standard, April 1995》，可参考 *http://www.cerberussystems.com/INFOSEC/stds/fip180-1.htm* 以及 GNU *coreutils* 包里的实例。

长的十六进制签名字符串只不过是一个具有许多位数的整数，它是由文件的所有字节计算得来，在这样的计算方式下，几乎不可能有任何其他字节字符串流能产生相同的值。使用好的算法，较长的签名一般来说就意味着较可能具有唯一性。md5sum输出32个十六进制数字，等同于128个位（注12）。因此两个不同的文件，要具有相同签名的可能性，大约为 $2^{64} = 1.84 \times 10^{19}$ 分之一，几乎小到微不足道。近期的密码学研究展现了建立一对具有相同MD5校验和的文件的可能。不过，产生一个与已存在文件内容类似但又不一致的文件，又要二者都具相同校验和，仍旧是一件困难的事。

为了在一组签名中找到相匹配的，使用它们作为签名计数表格里的索引，并且仅报告计数结果超过1的那些情况，awk正是可以帮我们完成它的工具，程序见例10-2。

例10-2：寻找匹配的文件内容

```
#! /bin/sh -
# 根据它们的 MD5 校验和，
# 显示在某种程度上内容几乎一致的文件名。
#
# 语法 ：
#       show-identical-files files

IFS='
	'

PATH=/usr/local/bin:/usr/bin:/bin
export PATH

md5sum "$@" /dev/null 2> /dev/null |
  awk '{
         count[$1]++
         if (count[$1] == 1) first[$1] = $0
         if (count[$1] == 2) print first[$1]
         if (count[$1] > 1)  print $0
       }' |
      sort |
          awk '{
                 if (last != $1) print ""
                 last = $1
                 print
               }'
```

下面是该程序在GNU/Linux系统下的输出结果：

```
$ show-identical-files /bin/*
```

注12：如果你从N个项目中选一个，则有1/N机会被选中。如果选M个项目，则有M(M-1)/2可能的配对，找到一个相同配对的机会是(M(M-1)/2)/N。针对M而言，该值到达可能性1/2约是N的平方根。这被称为生日悖论（birthday paradox）；你可以在有关密码学、数字理论、概率论书籍及相关网站中找到有用的信息，包括若干个经过验证的例子。

```
2df30875121b92767259e89282dd3002  /bin/ed
2df30875121b92767259e89282dd3002  /bin/red

43252d689938f4d6a513a2f571786aa1  /bin/awk
43252d689938f4d6a513a2f571786aa1  /bin/gawk
43252d689938f4d6a513a2f571786aa1  /bin/gawk-3.1.0

...
```

由本例可推论 `ed` 与 `red` 在此系统里为相同一致的程序，尽管根据它们被引用的名称，可能仍会产生不同的行为模式。

内容一致的文件多半是会彼此连接，特别是当这些文件出现在系统目录下时。`show-identical-files` 在应用到用户目录下时，可以提供更多有用的信息，因为用户目录下的文件不大可能是连接，比较可能是用户无意中做的副本。

10.7.4 数字签名验证

各种的校验和工具程序都提供单一数字，这是文件的特性，且几乎不可能与具有不同内容的一个文件的校验和相同。软件发布时，通常会包含分发文件的校验和，这可以让你方便得知所下载的文件是否与原始文件匹配。不过，单独的校验和不能提供验证（verification）工作：如果校验和被记录在你下载软件里的另一个文件中，则攻击者可以恶意地修改软件，然后只要相应地修订校验和即可。

这个问题的解决方案是公钥加密（public-key cryptography）。在这种机制下，数据的安全保障来自两个相关密钥的存在：一个私密密钥 —— 只有所有者知悉，以及一个公开密钥 —— 任何人都可得知。两个密钥的其中一个用以加密，另一个则用于解密。公开密钥加密的安全性，依赖已知的公开密钥及可被该密钥解密的文本，以提供一条没有实际用途的信息但可被用来恢复私密密钥。这一发明最大的突破是解决了一直以来密码学上极严重的问题：在需要彼此沟通的对象之间，如何安全地交换加密密钥。

私密密钥与公开密钥是如何使用和运行的呢？假设 Alice 想对一个公开文件签名，她可以使用她的私密密钥（private key）为文件加密。之后 Bob 再使用 Alice 的公开密钥（public key）将签名后的文件解密，这么一来即可确信该文件为 Alice 所签名，而 Alice 也无须泄露其私密密钥，就能让文件得到信任。

如果 Alice 想传送一份只有 Bob 能读的信给他，她应以 Bob 的公开密钥为信件加密，之后 Bob 再使用他的私密密钥将信件解密。只要 Bob 妥善保管其私密密钥，Alice 便可确信只有 Bob 能读取她的信件。

对整个信息加密其实是没有必要的：相对的，如果只有文件的校验和加密，它就等于有

数字签名（digital signature）了。如果信息本身是公开的，这种方法便相当有用，不过还需要有方法验证它的真实性。

在 GNU Privacy Guard（GnuPG，注 13）与 Pretty Good Privacy（PGP，注 14）里有相当多工具程序提供公开密钥加密机制。要完整说明这些包需要整本书才够，可到参考书目中的“安全性与密码学”部分寻找。然而，会使用它们就只为一个重要任务：数字签名（digital signature）的验证。我们仅在此说明GnuPG，因为它还在持续发展中，且构建较 PGP 简单，也适用于更多平台。

由于计算机越来越容易遭受攻击，许多的软件存档文件（archive）现在都并入文件校验和信息的数字签名，以及来自签名者的私密密钥。这也就是为什么了解验证这样的签名是很重要的原因，如果有签名文件，你应该都要记得验证它。使用GnuPG的方式如下：

```
$ ls -l coreutils-5.0.tar*                    显示分发文件
-rw-rw-r--  1 jones devel 6020616 Apr  2  2003 coreutils-5.0.tar.gz
-rw-rw-r--  1 jones devel      65 Apr  2  2003 coreutils-5.0.tar.gz.sig

$ gpg coreutils-5.0.tar.gz.sig                尝试验证此签名
gpg: Signature made Wed Apr  2 14:26:58 2003 MST using DSA key ID D333CBA1
gpg: Can't check signature: public key not found
```

签名验证失败，是因为我们还未将签名者的公开密钥加入gpg密钥环。如果我们知道谁对文件执行签名，我们就可以在签名者的个人网站上找到公开密钥，或是通过 email 向签名者要求一份密钥。然而，我们在这里拥有的就只有密钥ID的信息。幸好使用数字签名的人多半会将它们的公开密钥注册到第三方（thrid-party）的公开密钥服务器（public-key server），且该注册会自动地提供给其他的密钥服务器共享。几个主要的站点列于表 10-2，你也可以自查找引擎找到更多数据。你可以复制一份公开密钥，以提升安全性：如果密钥服务器不能用或毁损，便能轻松切换成另一个。

表 10-2：主要的公开密钥服务器

国家	URL
比利时	*http://www.keyserver.net/en/*
德国	*http://math-www.uni-paderborn.de/pgp/*
德国	*http://pgp.zdv.uni-mainz.de/keyserver/pks-commands.html#extract*
英国	*http://www.cl.cam.ac.uk/PGP/pks-commands.html#extract*
美国	*http://pgp.mit.edu/*

注 13： *ftp://ftp.gnupg.org/gcrypt/gnupg/* 与 *http://www.gnupg.org/*。

注 14： *http://web.mit.edu/network/pgp.html*。

在网页浏览器上造访这些密钥服务器，在查找栏里输入密钥ID 0xD333CBA1(前面的0x是强制性的)，并得到这样的报告：

```
Public Key Server -- Index ''0xD333CBA1 ''

Type bits /keyID    Date       User ID
pub  1024D/D333CBA1 1999/09/26 Jim Meyering <meyering@ascend.com>
...
```

按照密钥ID上的链接（例子的粗体字部分）可连到下面网页：

```
Public Key Server -- Get ''0xD333CBA1 ''

-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: PGP Key Server 0.9.6

mQGiBDftyYoRBACvICTt5AWe7kdbRtJ37IZ+ED5tBA/IbISfqUPO+HmL/J9JSfkV
QHbdQR5dj5mrU6BY5YOY7L4KOS61H3AgvsZ/NhkDBraBPgnMkpDqFb7z4keCIebb
...
-----END PGP PUBLIC KEY BLOCK-----
```

最后，将密钥内容存储到临时性文件，例 如temp.key，并加到你的密钥环中：

```
$ gpg --import temp.key                    将公开密钥，加到你的密钥环
gpg: key D333CBA1: public key "Jim Meyering <jim@meyering.net>" imported
gpg: Total number processed: 1
gpg:               imported: 1
```

现在，就可以成功地验证签名了：

```
$ gpg coreutils-5.0.tar.gz.sig             验证数字签名
gpg: Signature made Wed Apr  2 14:26:58 2003 MST using DSA key ID D333CBA1
gpg: Good signature from "Jim Meyering <jim@meyering.net>"
gpg:                     aka "Jim Meyering <meyering@na-net.ornl.gov>"
gpg:                     aka "Jim Meyering <meyering@pobox.com>"
gpg:                     aka "Jim Meyering <meyering@ascend.com>"
gpg:                     aka "Jim Meyering <meyering@lucent.com>"
gpg: checking the trustdb
gpg: checking at depth 0 signed=0 ot(-/q/n/m/f/u)=0/0/0/0/0/1
gpg: next trustdb check due at ????-??-??
gpg: WARNING: This key is not certified with a trusted signature!
gpg:          There is no indication that the signature belongs to the owner.
Primary key fingerprint: D70D 9D25 AF38 37A5 909A  4683 FDD2 DEAC D333 CBA1
```

成功验证中的警告信息简单扼要地告诉你：你仍未认证签名者密钥确实是属于他的。除非你私下认识签名者，并有很好的理由相信这个密钥是有效的，否则你不应认证此密钥。

攻击者可以修改再重新包装分发包，但不知道签名者的（秘密）私密密钥，数字签名不能被重新产生，且gpg会发现此攻击：

```
$ ls -l coreutils-5.0.tar.gz               列出遭恶意修改的存档文件
```

```
-rw-rw-r--  1 jones devel 6074205 Apr  2 2003 coreutils-5.0.tar.gz

$ gpg coreutils-5.0.tar.gz.sig                    试着验证数字签名
gpg: Signature made Wed Apr  2 14:26:58 2003 MST using DSA key ID D333CBA1
gpg: BAD signature from "Jim Meyering <jim@meyering.net>"
```

数字签名确保你站点里的文件匹配于远端站点中已准备妥且完成签名的那个文件。当然，当签名被验证时，在签名者系统上软件包装成包分发之前，就已经遭受到未侦测出的攻击是无法被显现出来的。安全性永远不可能是百分百完美。

你不一定要使用网页浏览器取得公开密钥：GNU的wget工具程序（注15）可以帮你完成这件事，前提是你必须先找出特定密钥服务器所预期的URL语法。例10-3的脚本可以让密钥的取得更容易，还会提醒你如何将公开密钥加入到你的密钥环中。

例10-3：自动化公开密钥的取得

```
#! /bin/sh -
# 自密钥服务器取得一个或多个 PGP/GPG 密钥
#
# 语法 :
#       getpubkey key-ID-1 key-ID-2 ...

IFS='
'

PATH=/usr/local/bin:/usr/bin:/bin
export PATH

for f in "$@"
do
  g=0x`echo $f | sed -e s'/^0x//'`                          确保字首为 0x
  tmpfile=/tmp/pgp-$g.tmp.$$
  wget -q -O - "http://pgp.mit.edu:11371/pks/lookup?op=get&search=$g" > $tmpfile
  ls -l $tmpfile
  echo "Try:      pgp -ka $tmpfile"
  echo "          pgpgpg -ka $tmpfile"
  echo "          rm -f $tmpfile"
done
```

使用范例如下：

```
$ getpubkey D333CBA1                    取得密钥 ID 为 D333CBA1 的公开密钥
-rw-rw-r--  1 jones jones 4567 Apr  6 07:26 /tmp/pgp-0xD333CBA1.tmp.21649
Try:    pgp -ka /tmp/pgp-0xD333CBA1.tmp.21643
        pgpgpg -ka /tmp/pgp-0xD333CBA1.tmp.21643
        rm -f /tmp/pgp-0xD333CBA1.tmp.21643
```

注15：　可在 *ftp://ftp.gnu.org/gnu/wget/* 取得。

一些密钥可同时用于PGP与GnuPG，但有很多是不可以的，所以提醒会包含两者。因为gpg与pgp的命令行选项各异，而pgp先开发，gpg则来自一个包装程序pgpgpg，其采用与pgp相同的选项，但却是调用gpg执行任务。在此pgpgpg -ka意同于gpg -import。

getpubkey可以将取得的密钥加入到你的GnuPG与/或PGP密钥环中，只要花点剪切/粘贴的气力。gpg则提供一次到位的方式，但它只更新你的GnuPG密钥环：

```
$ gpg --keyserver pgp.mit.edu --search-keys 0xD333CBA1
 gpg: searching for "0xD333CBA1" from HKP server pgp.mit.edu
Keys 1-6 of 6 for "0xD333CBA1"
(1)     Jim Meyering <meyering@ascend.com>
          1024 bit DSA key D333CBA1, created 1999-09-26
...
Enter number(s), N)ext, or Q)uit > 1
gpg: key D333CBA1: public key "Jim Meyering <jim@meyering.net>" imported
gpg: Total number processed: 1
gpg:               imported: 1
```

--keyserver选项只有第一次才需要，不过你之后还是可以用它来指定不同的服务器。除了密钥ID以外，--search-keys选项还可以接受电子邮件地址、用户名称或个人姓名。

10.8 小结

本章我们介绍的是如何使用ls与stat列出文件与文件meta数据，还有如何使用touch设置文件时间戳。touch可显示有关日期时间相关的信息以及在许多现行系统上的范围限制。

我们说明了如何以Shell的进程ID变量$$，搭配mktemp工具并自己动手取出随机数据流样本，建立唯一的临时性文件名称。计算机的世界可以说是一个充满敌意的环境，所以可通过此方式给予临时性文件具有唯一性与唯一访问性，让你的程序可以免于遭受攻击。

locate与slocate命令可用于定期更新的数据库（是经由完整地扫描文件系统所构建的）中，快速地查寻文件名称。当你知道全部或部分的文件名，且只想知道它在文件系统里的什么位置，那么使用locate就是最好的方式，除非文件是在查找数据库构建完成之后新产生的。

type命令是找出有关Shell命令相关信息的好方法，我们在第8章提供的pathfind脚本，则是提供较一般性的解决方案，便于找出特定目录路径下的文件。

我们花了很多篇幅探讨功能强大的 find 命令，它采用暴力破解遍历文件系统，寻找与用户指定条件匹配的文件。尽管如此，我们仍留下它许多未曾提及的性能，待你自行从使用手册或其他更好的 GNU find 文件里深入了解它。

我们简短说明了xargs的处理方式，这是另一个用以处理文件列表的命令，通常出现在上游为 find 的管道里。它除了能克服很多系统上命令行长度的限制，还能让你在管道里插入额外的过滤器，以便进一步处理文件。

df 与 du 命令会报告文件系统与目录树里的空间使用状态。把它们学好，因为你会经常用到它们。

最后，我们描述比较文件的命令、应用补丁、产生文件校验和以及验证数字签名。

第 11 章

扩展实例：合并用户数据库

到现在为止，我们已经一路学习、探索，也看过许多 Shell 脚本了。本章的目标是将前面所学的，结合 Shell 程序编写，挑战中等难度的任务。

11.1 问题描述

UNIX的密码文件 /etc/passwd已经在本书出现过很多次，系统管理者的工作多半也都是围绕着密码文件（还有相对的组文件 /etc/group）的操作。格式如下所示（注 1）：

```
tolstoy:x:2076:10:Leo Tolstoy:/home/tolstoy:/bin/bash
```

有7个字段：用户名称（username）、加密密码、用户ID编号（UID）、组ID编号（GID）、全名、根目录以及登录 Shell。字段为空不是个好做法：特别是第 2 个字段，如果为空，用户无须密码即可登录，且任何可以访问系统或其终端的人都可以该用户身份登录。如果第 7 个字段（Shell）为空，则 UNIX 默认为 Bourne Shell —— /bin/sh。

如我们在附录B里所讨论到的：用户与组ID编号，都为UNIX在访问文件时用来检查权限所用。如果两个用户具有不同的名称却拥有相同的UID编号，则就UNIX来说，它们是相同的（*identical*）。这种情况很少见，不过两个账号拥有相同UID编号是不对的。特别是NFS会要求一致的UID空间，用户编号2076在所有系统中通过NFS彼此访问，最好是相同的用户（tolstoy），否则有可能出现相当严重的安全性问题。

现在，随我们回到多年前（大约 1986 年）吧，那时 Sun 的 NFS 正越来越受欢迎，还能应用在非 Sun 的系统上。同时，我们之中有一个系统管理员，手下有两台为 4.2 BSD

注 1： BSD 系统还使用 /etc/master.passwd 文件，它具有三个额外的字段：用户的登录类别、密码变更时间以及账号过期时间。这些字段的位置就在 GID 字段与全名字段之间。

UNIX的计算机系统。这两个系统以TCP/IP相互通信，但未使用NFS。然而，新的OS厂商已规划要让4.3 BSD + NFS在这些系统上可使用。有许多用户在这两台系统上都有账号；基本上，用户名称都一样，但UID却不同!这些系统很快就要通过NFS共享文件系统；它们的UID空间要被合并是势在必行的。我们的任务就是编写一系列的脚本，功能是：

- 将两个系统里的 /etc/passwd文件合并。这是为了确保来自这两台系统的所有用户都具有独一无二的UID编号。
- 针对已存在的UID、但被应用在不同的用户身上的情况，则将其所有文件的所有权变更为正确用户。

这就是本章的任务，我们从零开始（原始的脚本太长，它只是偶尔的兴趣，并且像是在做学术研究）。这里的问题不单单是学术性的，试想：公司里有两个原本是分开的部门，现在是合并的时候，用户可能在多个部门的系统里都有账号。如果你是系统管理者，就有可能面临这样的任务。我们觉得解决这个问题应该是相当有趣的。

11.2 密码文件

我们就叫这两个假定的UNIX系统为u1与u2吧!例11-1呈现的是u1的/etc/passwd：

例 11-1：u1 的 /etc/passwd 文件

```
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
adm:x:3:4:adm:/var/adm:/sbin/nologin
tolstoy:x:2076:10:Leo Tolstoy:/home/tolstoy:/bin/bash
camus:x:112:10:Albert Camus:/home/camus:/bin/bash
jhancock:x:200:10:John Hancock:/home/jhancock:/bin/bash
ben:x:201:10:Ben Franklin:/home/ben:/bin/bash
abe:x:105:10:Honest Abe Lincoln:/home/abe:/bin/bash
dorothy:x:110:10:Dorothy Gale:/home/dorothy:/bin/bash
```

而例 11-2 为 u2 的 /etc/passwd：

例 11-2：u2 的 /etc/passwd 文件

```
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
adm:x:3:4:adm:/var/adm:/sbin/nologin
george:x:1100:10:George Washington:/home/george:/bin/bash
betsy:x:1110:10:Betsy Ross:/home/betsy:/bin/bash
jhancock:x:300:10:John Hancock:/home/jhancock:/bin/bash
```

```
ben:x:301:10:Ben Franklin:/home/ben:/bin/bash
tj:x:105:10:Thomas Jefferson:/home/tj:/bin/bash
toto:x:110:10:Toto Gale:/home/toto:/bin/bash
```

如果你仔细审视这些文件，就会发现我们程序所必须处理的可能情况很多：

- 用户在两个系统上都拥有相同的用户名称（username）与UID。这多半都是管理性账号，例如 root 和 bin。
- 用户的 username 与 UID 只有一台系统里有，另一台没有。这种情况在合并时，不会有问题。
- 用户在两台系统上拥有相同的 username，但 UID 不同。
- 用户在两台系统上拥有相同的 UID，但 username 不同。

11.3 合并密码文件

第一步就是先建立合并的 /etc/passwd 文件。这句话包含几个子步骤：

1. 直接地物理合并文件，将重复的 username 聚在一起，产生的结果将成为下个步骤的输入。
2. 将合并文件分割为三份，供而后处理：
   - 具相同 username 与 UID 的用户放进 unique1 文件。未重复的用户 username 也放入此文件。
   - 具相同 username，但不同 UID 的用户，放入第二个文件：dupusers。
   - 具相同 UID 但不同 username 的用户放入第三个文件：dupids。
3. 建立已使用中具唯一性的所有UID编号的列表。这是为了日后出现冲突而我们必须变更 UID 时（例如，用户 jhancock 与 ben），可用来寻找新的、未使用的 UID 编号。
4. 编写另一个程序，搭配使用中 UID 编号的列表，以便我们寻找新的、未使用的 UID 编号。
5. 建立用以产生最后 /etc/passwd记录的三项组合（username、旧的UID、新的UID）列表。还有最重要的：产生命令，以变更文件系统中文件的所有权。

   与此同时，针对原来就拥有数个 UID 的用户以及同一 UID 拥有多个用户，建立最后的密码文件项目。
6. 建立最终密码文件。

7. 建立变更文件所有权的命令列表，并执行它。这部分必须谨慎处理，有很多的地方必须小心规划。

另外，在这里提供的所有程序代码，前提假设都是在 username 与 UID 不会重复被使用超过两次之下运行的。实际上这不应该是个问题，但也值得你深思，将来有一天，你可能会遇到比这更复杂的情况。

11.3.1 根据管理性切分用户

合并密码文件不难，两个文件名分别为 u1.passwd 与 u2.passwd，我们以 sort 命令完成这件事，再搭配 tee 存储文件，并同时将其打印到标准输出以便能够看到：

```
$ sort u1.passwd u2.passwd | tee merge1
abe:x:105:10:Honest Abe Lincoln:/home/abe:/bin/bash
adm:x:3:4:adm:/var/adm:/sbin/nologin
adm:x:3:4:adm:/var/adm:/sbin/nologin
ben:x:201:10:Ben Franklin:/home/ben:/bin/bash
ben:x:301:10:Ben Franklin:/home/ben:/bin/bash
betsy:x:1110:10:Betsy Ross:/home/betsy:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
bin:x:1:1:bin:/bin:/sbin/nologin
camus:x:112:10:Albert Camus:/home/camus:/bin/bash
daemon:x:2:2:daemon:/sbin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
dorothy:x:110:10:Dorothy Gale:/home/dorothy:/bin/bash
george:x:1100:10:George Washington:/home/george:/bin/bash
jhancock:x:200:10:John Hancock:/home/jhancock:/bin/bash
jhancock:x:300:10:John Hancock:/home/jhancock:/bin/bash
root:x:0:0:root:/root:/bin/bash
root:x:0:0:root:/root:/bin/bash
tj:x:105:10:Thomas Jefferson:/home/tj:/bin/bash
tolstoy:x:2076:10:Leo Tolstoy:/home/tolstoy:/bin/bash
toto:x:110:10:Toto Gale:/home/toto:/bin/bash
```

例 11-3 呈现的是 splitout.awk，该脚本的功能是将合并后的文件切分为三个新文件名，分别为 dupusers、dupids 以及 unique1。

例 11-3：splitout.awk 程序

```
#! /bin/awk -f
#  $1    $2     $3  $4   $5          $6      $7
# user:passwd:uid:gid:long name:homedir:Shell

BEGIN { FS = ":" }

# name[]     --- 以 username 为索引
# uid[]      --- 以 uid 为索引

# 如果出现重复，决定其配置
```

```
{
    if ($1 in name) {
        if ($3 in uid)
            ;    # 名称与 uid 一致，什么事都不做
        else {
            print name[$1] > "dupusers"
            print $0 > "dupusers"
            delete name[$1]

            # 删除名称相同、uid 不同的已存储项目
            remove_uid_by_name($1)
        }
    } else if ($3 in uid) {
        # 知道 $1 并不在名称 name 里，所以存储重复的 ID 记录
        print uid[$3] > "dupids"
        print $0 > "dupids"
        delete uid[$3]

        # 删除具有相同 uid、不同名称的已存储项目
        remove_name_by_uid($3)
    } else
        name[$1] = uid[$3] = $0      # 第一次看到这条记录
}

END {
    for (i in name)
        print name[i] > "unique1"

    close("unique1")
    close("dupusers")
    close("dupids")
}

function remove_uid_by_name(n,     i, f)
{
    for (i in uid) {
        split(uid[i], f, ":")
        if (f[1] == n) {
            delete uid[i]
            break
        }
    }
}

function remove_name_by_uid(id,     i, f)
{
    for (i in name) {
        split(name[i], f, ":")
        if (f[3] == id) {
            delete name[i]
            break
        }
    }
}
```

程序的运行，是将每一输入行的副本保存在两个数组里，第一个数组以 username 为索引、第二个则以 UID 编号为索引。第一次看到一条记录时，username 与 UID 编号都不会存储到任何一个数组里，所以此行的副本会存储在这两者里。

当看到完全重复（username 与 UID 是相同一致）的记录时，不作任何事，因为我们已经有这个信息了。如果 username 已看到过而 UID 是新的，则两条记录都会写进 dupusers 文件里，且 uid 数组里的第一条记录副本会被删除，因为我们不再需要它了。类似的逻辑也应用在 UID 之前已看到过，但 username 不相符的记录上。

执行 END 规则时，所有留在 name 数组里的记录表示的是唯一的记录。它们会被写到 unique1 文件里，然后再关闭所有文件。

remove_uid_by_name() 与 remove_name_by_uid() 是 awk 函数。awk 里的用户定义函数在 9.8 节里已作过介绍。这两个函数的功能是分别自 uid 与 name 数组中，删除不再需要的信息。

执行建立这些文件的程序：

```
awk -f splitout.awk merge1
```

11.3.2 管理 UID

现在，我们已有分类的用户了，下一项任务，就是建立使用中的 UID 编号列表：

```
awk -F: '{ print $3 }' merge1 | sort -n -u > unique-ids
```

我们可以通过计算 merge1 与 unique-ids 里的行数，验证唯一的 UID 编号：

```
$ wc -l merge1 unique-ids
     20 merge1
     14 unique-ids
     34 total
```

继续我们的任务列表，下一步便是编写程序，产生未使用的 UID。默认的情况下，程序会读取使用中 UID 编号的排序后列表，然后打印第一个可用的 UID 编号。不过因为我们处理的是多用户，因此我们要的是一批未使用的 UID。这部分可使用 -c 选项指定，只要提供 UID 个数，即可应要求产生。例 11-4 的 newuids.sh 脚本便是执行这一任务。

例 11-4：newuids.sh 程序

```
#! /bin/sh -

# newuids --- 打印一个或多个未使用的 uid
#
# 语法 :
```

```
#   newuids [-c N] list-of-ids-file
#   -c N         显示 N 个未使用的 uid

count=1      # 预定要显示的 uid 个数

# 解析参数，令 sh 发出诊断
# 必要时离开程序
while getopts "c:" opt
do
    case $opt in
    c)  count=$OPTARG ;;
    esac
done

shift $(($OPTIND - 1))

IDFILE=$1

awk -v count=$count '
    BEGIN {
        for (i = 1; getline id > 0; i++)
            uidlist[i] = id

        totalids = i

        for (i = 2; i <= totalids; i++) {
            if (uidlist[i-1] != uidlist[i]) {
                for (j = uidlist[i-1] + 1; j < uidlist[i]; j++) {
                    print j
                    if (--count == 0)
                        exit
                }
            }
        }
    }' $IDFILE
```

绝大多数的工作都是在awk行内程序中完成。首先读取UID编号列表到uidlist数组中，for循环处理整个数组。当它发现两个元素值不相邻时，则依次通过并显示在那些元素之间的内含值，每次减少count，使得只有count个UID编号会显示。

在Shell里，更能直接作数组处理与算术的支持，例如ksh93与bash，让Shell做完所有的工作并不是不可能。事实上，这个awk脚本乃派生自ksh93的某个产物：见 *http://linux.oreillynet.com/pub/a/linux/2002/05/09/uid.html*。

11.3.3 用户－旧UID－新UID的建立

现在要处理的是dupusers与dupids这两个文件。输出文件将列出以空格作为分隔、一行一条记录的“username－旧UID－新UID”列表，以便再作处理。对dupusers来说，处理方式很直接明了：第一条遇到的记录为旧UID，下一个则是新选择的UID（换

句话说，我们任意地决定使用第二个较大的UID供用户的所有文件使用）。与此同时，我们还为列在这两者文件中的用户建立最终的 /etc/passwd 记录。

注意： 这个计划平等对待两个系统的磁盘，要求两个系统上的文件所有权都需变更。对于可能耗费更多时间改变文件所有权来说，编写这个程序代码是简单多了。另一种选择是保持某个系统里的文件不动，我们称该系统为主系统，也就是说只改变第二台系统里的所有权。这部分的程序就不好写了，我们把这部分留给读者自行练习。

这是程序代码：

```
rm -f old-new-list

old_ifs=$IFS
IFS=:
while read user passwd uid gid fullname homedir Shell
do
    if read user2 passwd2 uid2 gid2 fullname2 homedir2 Shell2
    then
        if [ $user = $user2 ]
        then
            printf "%s\t%s\t%s\n" $user $uid $uid2 >> old-new-list
            echo "$user:$passwd:$uid2:$gid:$fullname:$homedir:$Shell"
        else
            echo $0: out of sync: $user and $user2 >&2
            exit 1
        fi
    else
        echo $0: no duplicate for $user >&2
        exit 1
    fi
done < dupusers > unique2
IFS=$old_ifs
```

我们使用 Shell 的 read 命令，自 dupusers 读取每一组行，传送最终密码文件记录予 unique2。同时，将想要的输出传至新文件 old-new-list。这里必须使用 >> 运算符，因为我们是使用循环，每次加入一条新记录。为确保该文件为全新状态，在进入循环主体之前先作删除操作。

设置 IFS 为：，可以让密码文件行的读取更容易，它能够正确地处理每条以冒号隔开的字段。IFS 的原始值存储在 old_ifs 内，并在循环之后被恢复（我们当然可以直接使用 IFS= :read... 的方式，但这么做我们在处理两个 read 语句时就得更小心了）。

类似的程序代码也可应用于UID编号相同、username不同的用户。这里我们一样选择简化：给所有这样的用户一个全新的、未使用的 UID 编号。（也就是说，让每组里的第一

个用户保持原有的UID编号；然而这需要只变更第二个用户所在的系统里的文件所有权。在现实情况下，这么做的确比较好）。

```
count=$(wc -l < dupids)       # 计算所有重复的id

# 如果 POSIX sh 里有数组，请这么用：
set -- $(newuids.sh -c $count unique-ids)

IFS=:
while read user passwd uid gid fullname homedir Shell
do
    newuid=$1
    shift

    echo "$user:$passwd:$newuid:$gid:$fullname:$homedir:$Shell"

    printf "%s\t%s\t%s\n" $user $uid $newuid >> old-new-list
done < dupids > unique3
IFS=$old_ifs
```

为了更方便拥有所有新的UID编号，我们通过set与命令替换功能，将它们放置在位置参数中。然后通过从$1开始指定，可自循环中将每个新UID取出，并使用shift将下一个替换。完成时，我们将拥有三个新的输出文件：

```
$ cat unique2                           拥有两个 UID 的用户
ben:x:301:10:Ben Franklin:/home/ben:/bin/bash
jhancock:x:300:10:John Hancock:/home/jhancock:/bin/bash

$ cat unique3                           取得新 UID 的用户
abe:x:4:10:Honest Abe Lincoln:/home/abe:/bin/bash
tj:x:5:10:Thomas Jefferson:/home/tj:/bin/bash
dorothy:x:6:10:Dorothy Gale:/home/dorothy:/bin/bash
toto:x:7:10:Toto Gale:/home/toto:/bin/bash

$ cat old-new-list                      用户-old-new 列表
ben      201      301
jhancock          200      300
abe      105      4                     见下个段落说明
tj       105      5
dorothy  110      6
toto     110      7
```

最后的密码文件乃由三个unique? 文件合并而成。虽然cat可完成此工作，不过以UID编号顺序合并它们会是比较好的方式：

```
sort -k 3 -t : -n unique[123] > final.password
```

通配字符unique[123]展开为三个文件名unique1、unique2以及unique3。以下为排序的最后结果：

```
$ cat final.password
root:x:0:0:root:/root:/bin/bash
```

```
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
adm:x:3:4:adm:/var/adm:/sbin/nologin
abe:x:4:10:Honest Abe Lincoln:/home/abe:/bin/bash
tj:x:5:10:Thomas Jefferson:/home/tj:/bin/bash
dorothy:x:6:10:Dorothy Gale:/home/dorothy:/bin/bash
toto:x:7:10:Toto Gale:/home/toto:/bin/bash
camus:x:112:10:Albert Camus:/home/camus:/bin/bash
jhancock:x:300:10:John Hancock:/home/jhancock:/bin/bash
ben:x:301:10:Ben Franklin:/home/ben:/bin/bash
george:x:1100:10:George Washington:/home/george:/bin/bash
betsy:x:1110:10:Betsy Ross:/home/betsy:/bin/bash
tolstoy:x:2076:10:Leo Tolstoy:/home/tolstoy:/bin/bash
```

11.4 改变文件所有权

乍看之下，改变文件所有权很简单。只要提供 username 与新 UID 编号列表，我们应该能编写一个像下面这样的循环（需以 root 权限执行）：

```
while read user old new
do
        cd /home/$user                    改变用户目录
        chown -R $new .                   递归地改变所有权，见 chown(1)
done < old-new-list
```

这个程序的想法是：改变用户的根目录，并递归执行 chown，将所有文件、目录都改成新的 UID 编号。不过，这是不够的！用户的文件有可能放在根目录以外的地方。举例来说，有两个用户 ben 与 jhancock，它们共同参与一个项目，置于 /home/ben/declaration 下：

```
$ cd /home/ben/declaration
$ ls -l draft*
-rw-r--r--    1 ben         fathers     2102 Jul  3 16:00 draft10
-rw-r--r--    1 jhancock    fathers     2191 Jul  3 17:09 draft.final
```

如果我们只是作递归的 chown 处理，两个文件最后都会属于 ben，而 jhancock 在大文件系统重组织（Great Filesystem Reorganization）过后，不会高兴把每日的工作归功给 ben 的。

不过更糟的情况应该是：用户拥有的文件，放在根目录之外的地方。/tmp 就是一个明显的例子，不过还有源代码管理系统，像 CVS 也会有这种情况。CVS 针对项目存储主文件（master files）在软件库内，通常这个地方不会是任何人的根目录，且多半是在系统目录的某一处。软件库的原始文件属于多位用户。这些文件的所有权也应该作更改。

这样，确保所有文件在每个地方都被正确更改的唯一方式，便是使用 find，从根目录开始做。完成此目标最显而易见的方式就是在 find 里执行 chown，像这样：

```
find / -user $user -exec chown $newuid '{}' \;
```

这么做会执行彻底的文件查找，检查系统里每一个文件与目录，看是否有属于$user用户的东西。find会针对每个相符的文件与目录执行chown，将所有权改为$newuid里的UID（find命令在10.4.3节有说明，-exec选项会针对每一个与条件比对相符的文件执行接下来的所有参数，直至分号为止。find命令里的{}意指替换找到的文件名称至命令中）。不过，这种使用find的代价很高，因为它会针对每一个文件或目录，建立一个新的chown进程。因此，我们结合find与xargs：

```
# 一般版本 :
find / -user $user -print | xargs chown $newuid

# 如果你有GNU工具集：
# find / -user $user -print0 | xargs --null chown $newuid
```

这样做执行一样彻底的文件查找，这次会打印系统里每个属于$user的文件与目录的名称。该列表之后会以管道传递给xargs，它会尽可能地在所有文件上执行chown，将所有权改为$newuid里的UID。

现在，考虑old-new-list文件里出现这样数据的情况：

```
juser          25      10
mrwizard       10      30
```

这里有顺序的问题。如果我们在变更mrwizard文件的所有权之前，先改变所有juser的文件为UID 10，则所有juser的文件最后将会变成mrwizard所拥有！

这部分可使用UNIX的tsort程序解决，该程序为拓扑排序（Topological sorting针对部分已排序的数据，强化完整排序功能）。以我们的目标来看，必须以新UID、旧UID的顺序，将数据传给tsort：

```
$ tsort << EOF
> 30 10
> 10 25
> EOF
30
10
25
```

输出结果告诉我们：必须在25改变为10之前，将10改为30。你应该可以想象，必须非常小心写脚本。不过，我们可以做一些巧妙的处理，完全避开这个问题！记得不同名称下拥有重复的UID编号的情况吗？

```
$ cat dupids
abe:x:105:10:Honest Abe Lincoln:/home/abe:/bin/bash
tj:x:105:10:Thomas Jefferson:/home/tj:/bin/bash
```

```
dorothy:x:110:10:Dorothy Gale:/home/dorothy:/bin/bash
toto:x:110:10:Toto Gale:/home/toto:/bin/bash
```

我们给所有这些用户一个全新的UID：

```
$ cat final.passwd
...
abe:x:4:10:Honest Abe Lincoln:/home/abe:/bin/bash
tj:x:5:10:Thomas Jefferson:/home/tj:/bin/bash
dorothy:x:6:10:Dorothy Gale:/home/dorothy:/bin/bash
toto:x:7:10:Toto Gale:/home/toto:/bin/bash
...
```

提供它们在任何地方都没使用的UID编号，就无须担心 find 命令的顺序了。

主程序的最后部分是产生find与xargs命令的列表。我们选择将该命令列表写到文件chown-files中，这么一来便能分别在后台中执行。这是由于程序执行极可能耗费许多时间，而我们的系统管理员，在花了这么多时间开发与测试这个脚本后，开始执行它之后应该想要好好回家睡个大觉了！脚本的最后结果如下：

```
while read user old new
do
    echo "find / -user $user -print | xargs chown $new"
done < old-new-list > chown-files

chmod +x chown-files

rm merge1 unique[123] dupusers dupids unique-ids old-new-list
```

这里，chown-files文件看起来如下所示：

```
$ cat chown-files
find / -user ben -print | xargs chown 301
find / -user jhancock -print | xargs chown 300
find / -user abe -print | xargs chown 4
find / -user tj -print | xargs chown 5
find / -user dorothy -print | xargs chown 6
find / -user toto -print | xargs chown 7
```

还记得 old-new-list 文件吗？

```
$ cat old-new-list
ben      201      301
jhancock          200      300
abe      105      4
tj       105      5
dorothy  110      6
toto     110      7
```

你可能已经注意到abe与tj两者一开始拥有相同UID，类似情况也出现在dorothy与toto上。执行chown-files时发生什么事了？不是所有tj的文件最后都属于新的UID

4吗？不是所有toto的文件，最后都属于新的UID 6吗？我们不是做好避开问题的措施了吗？

答案是：在放置新的/etc/passwd文件到每个系统上之前，只要我们分别在每个系统上执行这些命令，就会是很安全的。记得一开始，abe与dorothy只存在于u1里，而tj与toto也只在u2里，因此，当chown-files搭配原来的/etc/passwd在u1里执行时，find都不会寻找tj或toto的文件，因为这些用户不存在：

```
$ find / -user toto -print
find: invalid argument `toto' to `-user'
```

类似失败的情况也会出现在u2的对照组里。完整的merge-systems.sh脚本如例11-5所示。

例11-5：merge-systems.sh程序

```
#! /bin/sh

sort u1.passwd u2.passwd > merge1

awk -f splitout.awk merge1

awk -F: '{ print $3 }' merge1 | sort -n -u > unique-ids

rm -f old-new-list

old_ifs=$IFS
IFS=:
while read user passwd uid gid fullname homedir Shell
do
    if read user2 passwd2 uid2 gid2 fullname2 homedir2 Shell2
    then
        if [ $user = $user2 ]
        then
            printf "%s\t%s\t%s\n" $user $uid $uid2 >> old-new-list
            echo "$user:$passwd:$uid2:$gid:$fullname:$homedir:$Shell"
        else
            echo $0: out of sync: $user and $user2 >&2
            exit 1
        fi
    else
        echo $0: no duplicate for $user >&2
        exit 1
    fi
done < dupusers > unique2
IFS=$old_ifs

count=$(wc -l < dupids)       # 计算重复的 id 数目

# 如果 POSIX sh 有数组，请这么用：
set -- $(newuids.sh -c $count unique-ids)
```

```
IFS=:
while read user passwd uid gid fullname homedir Shell
do
    newuid=$1
    shift

    echo "$user:$passwd:$newuid:$gid:$fullname:$homedir:$Shell"

    printf "%s\t%s\t%s\n" $user $uid $newuid >> old-new-list
done < dupids > unique3
IFS=$old_ifs

sort -k 3 -t : -n unique[123] > final.password

while read user old new
do
    echo "find / -user $user -print | xargs chown $new"
done < old-new-list > chown-files

chmod +x chown-files

rm merge1 unique[123] dupusers dupids unique-ids old-new-list
```

11.5 其他真实世界的议题

还有其他真实世界里可能面临的议题。在这里我们不写程序代码，仅作简短的讨论。

首先最明显的是 /etc/group 文件也可能得合并。针对此文件来说，必须做的有：

- 确认合并后的 /etc/group 已包含所有来自个别系统里的所有组，且具有相同的唯一 GID。这几乎完全与我们解答过的 username/UID 议题相似，只是文件格式有所不同。
- 在不同系统上的相同组中，进行用户的逻辑性合并。例如：

  ```
  floppy:x:5:tolstoy,camus                在 u1 /etc/group 中
  floppy:x:5:george,betsy                 在 u2 /etc/group 中
  ```

  当文件被合并时，组 floppy 的项目必须是：

  ```
  floppy:x:5:tolstoy,camus,george,betsy   用户的顺序不重要
  ```

- 所有文件的 GID 必须与合并后的新 /etc/group 同步，就像 UID 的处理一样。如果你够聪明，应该知道要产生一个包含 UID 与 GID 的 find … | xargs chown … 命令，让它们只要被执行一次就好。可节省机器处理时间，但要花费额外写程序的时间。

再者，任何一个大型的系统，都可能会出现文件拥有已不存在于 /etc/passwd 与 /etc/group 里的 UID 或 GID 值。要寻找这类的文件，可以这么做：

```
find / '(' -nouser -o -nogroup ')' -ls
```

这样做将产生类似 ls-dils 输出格式的文件列表。这类列表可能应该要做人工检查，来决定哪些用户与/或组应重新指定，或者需建立哪些新的用户（与/或组）。

以前者来说，可将文件再进一步处理：产生 find... | xargs chown...这样的命令完成任务。

后者只是简单地将对应的 UID 与 GID 名称，加入到 /etc/passwd 与 /etc/group 文件中，不过你应特别留意这些未使用的 UID 与 GID 编号，是否未与合并所产生的 UID 与 GID 冲突。如果你在合并之前建立这些新用户与组名称，就不会遇到冲突问题。

第三，在改变文件的用户与组处理期间，文件系统绝对得静止。即，处理时不应有任何其他活动发生。最好是让系统执行在单用户模式（single-user mode）下，只有超级用户 root 可以登录，且只能在系统的物理 console 设备上完成此任务。

最后可能就是效率议题了。来看看之前呈现的一连串命令：

```
find / -user ben -print | xargs chown 301
find / -user jhancock -print | xargs chown 300
...
```

这些管道的每一个，都会将计算机里的所有文件找过一遍，处理每个必须变更 UID 或 GID 的操作。在用户很少或系统文件不多的情况下（像是只有一个硬盘的系统），这还可以忍受。但如果有几百或几千个用户的文件必须更改，或是系统拥有许多个非常大的磁盘，我们就得使用另一个解决方案。使用像这样的管道：

```
find / -ls | awk -f make-commands.awk old-to-new.txt - > /tmp/commands.sh
... 在执行它之前先检查 /tmp/commands.sh ...
sh /tmp/commands.sh
```

这里的 make-commands.awk 命令为 awk 程序，先读取来自 old-to-new.txt 的旧换新 UID 变更（此文件可通过修改本章先前提过的脚本产生）。然后，make-commands.awk 会针对每一个输出的文件寻找是否有必需被变更的用户。如果是如此，则显示 chown 命令列。一旦所有的命令都被存储，便能在执行它们之前先看过（这部分我们一样还是留给读者作为自我练习）。

11.6 小结

本章已经重新建立并解决真实世界会遇到的问题：将两个分开的计算机系统里的密码文件合并，以便通过 NFS 共享它们的文件。

经过对密码文件的细心研究，我们可以将用户归类为两种：只在第一个系统里或只在第二个系统里，以及两个系统里都有的用户。问题在于我们必须确保每个用户，在两个系统里拥有一致且具有独一无二的 UID 编号，而每个用户的文件也都只属于他们自己。

解决此问题需要寻找新的未使用UID编号，以便出现UID冲突时可取用之，而且还必须留意改变文件所有权的命令顺序。这两个系统必须彻底查找，确定每一个文件的拥有者都已正确更新。

其他需要解决的议题，在形式上其实差不多：最显著的应该是合并组文件，以及将所有无人认领的文件指定拥有者。为安全起见，当这些运行在进行中时，系统应该是静止无任何活动的状态，我们也大致说明了在效率前提下的另一个解决方案。

解决方案包含了针对原始密码文件进行谨慎的过滤，使用了 awk、sort、uniq，以及大量使用 while read... 循环处理数据、准备改变用户文件所有权的命令。以 find、xargs，以及 chown（当然）完成此任务。

整个解决方案的程序代码不到170行，这数字还包含了注释！以C程序解决相同问题可能不只是产生更庞大的程序代码，可能编写、测试与除虫会耗掉更多的时间。而我们的解决方案，通过个别执行的命令提供更安全的做法，因为提供进行人为检查的机会，在变更文件所有权之前先作确认。我们认为这是一个相当详尽的示范，让读者们了解UNIX工具集的强大功能，并了解如何通过软件工具（Software Tool）解决手边的问题。

第 12 章

拼写检查

本章利用拼写检查，呈现各种Shell脚本的不同方面。介绍完spell程序后，我们会告诉你如何构建一个简单又好用的拼写检查程序。紧接着再介绍如何利用这个简单的Shell脚本，修改手边两个可自由使用的拼写程序的输出，让它看起来就像传统UNIX的spell程序那样。最后，呈现awk写成的拼写检查程序，让读者完整了解这个语言的简单利落。

12.1 spell 程序

spell程序做的事就是你想的：检查文件里是否有拼写错误。这个程序会读取命令行上指定的所有文件，在标准输出上产生排序后的单词列表，这个列表上的单词不是在它的字典里找不到，就是无法从标准的英文文法应用里派生出来（例如“words”派生自“word”）。有趣的是：POSIX 并未对 spell 进行标准化，在它的文件里是这么说的：

> 该工具程序对 Shell 脚本或传统应用程序并无用处。spell 是深思熟虑后的设计，但它却忽略了：用户指定的输入如果未伴随完整的字典，则没有技术可识别用户指定的输入文件。

我们不同意上述的第一部分。试想脚本的自动调试与问题报告，有人可能会想要显示类似的这些行：

```
#! /bin/sh -

# probreport --- 简单的问题报告程序

file=/tmp/report.$$
echo "Type in the problem, finish with Control-D."
cat > $file

while true
```

```
do
    printf "[E]dit, Spell [C]heck, [S]end, or [A]bort: "
    read choice
    case $choice in
    [Ee]*)  ${EDITOR:-vi} $file
            ;;
    [Cc]*)  spell $file
            ;;
    [Aa]*)  exit 0
            ;;
    [Ss]*)  break    # from loop
            ;;
    esac
done
...                 传送报告
```

在本章，我们会从各个角度审视拼写检查，因为这部分很有趣，我们有机会以不同的方式解决问题。

12.2 最初的UNIX拼写检查原型

以拼写检查为主题的研究报告及书籍已经不少于300项（注1）。在Jon Bentley的《Programming Pearls》一书（注2）里曾提到：Steve Johnson在1975年的某个午后，写出第一版spell。Bentley之后略为改造，贡献给Kernighan与Plauger（注3），该程序以UNIX的管道完成，我们可用现代语汇改写如下：

```
prepare filename |                          删除格式化命令
  tr A-Z a-z |                              将大写字母对应到小写字母
    tr -c a-z '\n' |                        删除标点符号
      sort |                                将单词依字母顺序排列
        uniq |                              删除重复的单词
          comm -13 dictionary -             报告不在字典里的单词
```

这里的*prepare*为过滤程序，它会将所有文件标记（markup）拿掉；最简单的情况，只要用到cat。我们使用的参数语法是假定tr命令为GNU版本。

这个管道里唯一我们还未曾提过的程序便是comm：它是用以比较两个排序后的文件，并选定或拒绝两个文件里共同的行。这里使用-13选项，因此它仅输出来自第二个文件（管道的输入）但不在第一个文件（字典）里的行。该输出为拼写异常报告。

注1： *http://www.math.utah.edu/pub/tex/bib/index-table-s.html#spell* 提供了丰富的参考文献。

注2： Jon Louis Bentley，《Programming Pearls》，Addison-Wesley，1986，ISBN 0-201-10331-1。

注3： Brian W. Kernighan and P.J.Plauger，《Software Tools in Pascal》，Addison-Wesley，1981，ISBN 0-201-10342-7。

comm

语法

```
comm [ options ... ] file1 file2
```

用途

指出在两个输入文件里，哪些行是只出现在其中一个文件中，或者两个文件里都有出现。

主要选项

-1

不要显示第一列（只在 *file*1 出现的行）

-2

不要显示第二列（只在 *file*2 出现的行）

-3

不要显示第三列（两个文件里都有的行）

行为模式

逐行读取两个文件，且输入文件必须都已排序。产生的输出共三列：只在 *file*1 里有的行、只在 *file*2 里有的行，以及两个文件里都有的行列。这两个文件名其中一个可以是 -，指定 comm 读取标准输入。

警告

它的选项不是直接式的；用户很难记得为了删除一个输出列，要加上一个选项！

Bentley之后继续讨论贝尔实验室的Doug McIlroy于1981年所开发的拼写检查程序，包括它的设计与应用、如何将字典存储在小型的内存里，以及为什么检查拼写这么困难，尤其是在像英文这样杂乱无章的语言上。

现代的 spell 为了效率而使用C完成。不过，原始的管道仍在贝尔实验室里使用相当长的一段时间。

12.3 改良的ispell与aspell

UNIX的 spell 支持许多选项，不过有很多是平时用不到的。不过令 spell 的行为模式偏向英式拼法的 -b 选项异常：它会以“centre”取代“center”、以“colour”取代“color”等（注4）。要了解其他选项请见使用手册。

注4： *spell*(1)使用手册中的BUGS小节，有长篇说明“British spelling was done by an American”。

其中一个不错的功能就是：你可以提供你本地有效单词的拼写列表。例如，在特定专门领域中的正确拼法，但在spell的字典里是不存在的（例如：POSIX）。你可以建立并长期维护自有的有效、但非一般性的单词列表，然后在执行spell时使用此列表。指定本地拼写列表的方式，是标明路径名称并将它放在要被检查的文件之前，且前置单个的+字符：

```
spell +/usr/local/lib/local.words myfile > myfile.errs
```

12.3.1 私有拼写字典

我们觉得，在实际上最重要的就是：针对你所写的任何文件都提供私有拼写字典。一个通用于大部分文件的字典并不实用，因为词汇量会变得太大，且无法确切发现错误。"syzygy"在数学论文里可能是正确的，但在小说里，它或许应为"soggy"才对。我们发现，几百万行的技术文件大全配合拼写字典作比较，几乎是每六行就出现一个拼写异常。这告诉我们，拼写异常很常见，这也是这个项目另外必须解决的问题。

关于spell还有一些棘手的事：它只能有一个+选项，且其字典必须以字典编纂法的方式排序，这是个很粗糙的设计。即spell的绝大多数版本，在locale变动后，就会失灵（虽然这被认为是个很差的设计，但事实上那只是未预期到locale的出现所导致的结果。spell的代码在很多系统上已使用20年以上未作更改，而当底层的程序库被更新以完成locale为主的排序时，没有人了解这会造成影响）。举例如下：

```
$ env LC_ALL=en_GB spell +ibmsysj.sok < ibmsysj.bib | wc -l
  3674
$ env LC_ALL=en_US spell +ibmsysj.sok < ibmsysj.bib | wc -l
  3685
$ env LC_ALL=C spell +ibmsysj.sok < ibmsysj.bib | wc -l
  2163
```

然而，如果私人字典的排序符合当前locale环境，则spell可适当运行：

```
$ env LC_ALL=en_GB sort ibmsysj.sok > /tmp/foo.en_GB
$ env LC_ALL=en_GB spell +/tmp/foo.en_GB < ibmsysj.bib | wc -l
  2163
```

问题是默认的locale在操作系统版本之间可能有所不同。因此最好的方式便是将LC_ALL环境变量设置为与私人字典排序一致，再执行spell。我们将在下一节提供spell已排序字典需求的改写方案。

12.3.2 ispell 与 aspell

这里有两个可以自由取用的拼写检查程序：ispell与aspell。ispell程序为交互模式的拼写检查，它会显示文件，然后将所有拼写错误之处反白，并提供建议的更动。aspell

程序也类似，不过它在英文上提供的建议更正比较好，且其作者希望它最终能取代 ispell。这两个程序都能用于产生拼错单词的简易列表，且因为希望 apsell 能取代 ispell，所以它们使用的选项相同：

-l
: 在标准输出打印拼错的单词列表。

-p *file*
: 以 *file* 作为正确单词拼法的个人字典。类似 UNIX spell 里，以 + 起始的私有文件选项。

ispell 的官方网站在 *http://ficus-www.cs.ucla.edu/geoff/ispell.html*，其源代码可在 *ftp://ftp.gnu.org/gnu/non-gnu/ispell/*（注 5）找到。aspell 官方网站则为 *http://aspell.net/*，源代码位于 *ftp://ftp.gnu.org/gnu/aspell/*。

这两个程序都提供基本的批处理拼写检查功能。当然它们同样也共享了坏习惯：产生未排序的结果，且未省略错误单词重复的部分（UNIX 的 spell 没有这两个问题）。因此，我们优秀的 GNU/Linux 厂商便有了 /usr/bin/spell 这样的 Shell 脚本出现：

```
#!/bin/sh

# aspell -l 大概地模仿标准 UNIX spell 程序

cat "$@" | aspell -l --mode=none | sort -u
```

--mode 选项使得 aspell 忽略一些类型的标记，例如 SGML 与 TEX。这里的 --mode=none 表示不做任何的过滤。sort -u 命令则是将排序结果里重复的部分去除，产生 UNIX 老手预期看到的结果。你也可以使用 ispell 作同样的事：

```
cat "$@" | ispell -l | sort -u
```

有两种方式可以再改进这个脚本，让它可以提供个人字典，像 UNIX 的 spell 那样。第一个替换 spell 脚本的方式如例 12-1。

例 12-1：以 ispell 代替 spell

```
#!/bin/sh

# UNIX的spell把`+file'的第一个参数看作是
# 提供私有拼写列表，我们也如法泡制。

mydict=
case $1 in
```

注 5： emacs 利用 ispell 在交互模式下进行拼写检查，处理速度会很快，因为 ispell 会在后台中持续执行。

```
+?*)    mydict=${1#+}    # 去除开头的 +
        mydict="-p $mydict"
        shift
        ;;
esac

cat "$@" | ispell -l $mydict | sort -u
```

这段代码只是查找起始为+的第一个参数，将其存储到变量，截去+字符，再放入准备好的 -p 选项、传递给 ispell 引用。

不幸的是：该技巧在 aspell 下无效，因为它要求字典必须为编译后的二进制格式。如要使用 aspell，我们改以 fgrep 手段进行，它可以匹配文件内所提供的多个字符串。我们另加入 -v 选项，要求 fgrep 显示不匹配的行。所以第二种替换 spell 脚本的方式见例 12-2。

例 12-2：以 aspell 取代 spell

```
#!/bin/sh

# UNIX的spell把`+file'的第一个参数看作是
# 私有拼写列表的提供，我们也如法泡制。

mydict=cat
case $1 in
+?*)    mydict=${1#+}    # 去除开头的 +
        mydict="fgrep -v -f $mydict"
        shift
        ;;
esac

# aspell -l 模仿标准 UNIX spell 程序。

cat "$@" | aspell -l --mode=none | sort -u | eval $mydict
```

如果你不想要排序私有字典，或不想烦恼不同的locale所产生的不同排序方式，那么相同的 fgrep 后续处理技巧也可搭配 UNIX 的 spell 使用。

下一节呈现的是 spell 的 awk 版本，它可以提供功能强大又简化的替代方案，是我们在此所讨论的 spell 替换的另一种选择。

12.4 在 awk 内的拼写检查程序

本节要呈现的是提供检查拼写的程序。即便所有UNIX系统都有 spell，有些甚至还有 aspell或ispell，但我们的程序兼具了教育性与实用性。本节不但能让你知道awk的功能有多强大，还能得到一个适用于所有平台上的程序，只要它有 awk。

我们必须强调检查（checking）与更正（correcting）的差别。后者必须了解内文格式，还需要人为确认，因此完全不适于批处理处理。由网页浏览器与文本处理程序所提供的自动化拼写更正只会让情况变得更糟，因为它们多半是错的，且在你快速输入时进行的二次推断，情况只会进一步恶化。

emacs文字编辑程序提供三个好的解决方案，可在输入内文期间提供拼写协助：可依需求展开部分单词以动态补齐单词、通过单一按键提出对当前单词的拼写验证需求，以及flyspell 程序库可用以要求以不那么显眼的颜色标示出可能有误的单词。

只要你能在拼写程序指出错误时认得拼错的部分，那么能够报告可能拼错单词的列表及允许你提供个人的特殊单词列表，非字典的一般单词，会是比较好的拼写检查程序，可减少该报告长度。之后你可以利用这份报告识别错误的部分，修正它们后再重新产生报告（这时应只有正确的单词了），然后将它的内容加入到你私有字典里。由于我们的写作都在处理技术的素材，它时常充满不常见的单词，实际上我们保有私人的与特定文件的补充字典，应用到我们所写的每个文件中。

为引导程序的进行，这里列出几个我们的拼写检查程序预期的设计目标。遵循ISO标准的实际，我们使用将会（shall）指出必须做的，而使用应该（should）指出想要做的：

- 程序将会能够读取文字数据流、隔离单词，以及报告不在已知单词列表［也即，拼写字典（spelling dictionary）］里的单词实体。
- 将会有一个默认的单词列表，由一个或多个系统字典收集而成。
- 它将可能取代默认的单词列表。
- 标准单词列表将有可能由一个或多个用户所提供的单词列表而扩增。该列表在技术性文件上特别有用，例如首字母缩写、术语及专有名词，它们大部分都无法在标准列表里找到。
- 单词列表将无须排序，这点与UNIX的spell不同，后者当locale变动时，会出现不当的行为模式。
- 虽然默认单词列表都是英文，但辅以适当的替代性单词列表，程序将可以处理任何语言的文字，只要它是以基础为ASCII的字符集（编码为8位字节）呈现，以空白字符（whitespace）分隔单词。这消除了难懂语言的难度，例如老挝语（Lao）与泰文（Thai），它们缺乏单词内的空间，因此需要更具扩展性的语意分析才能识别单词。
- 将忽略字母大小写，让单词列表维持在易于管理的大小，不过异常列表报告时将使用原来的大小写。

- 将忽略标点符号与数字，但顿点符号（缩写的一撇）将视为字母。
- 默认的报告将为排序后具有独一无二单词的列表(是无法在结合的单词列表里找到的）以一行一个单词的方式呈现。这是拼写异常列表（spelling exception list)。
- 将可通过选项增加异常列表报告，并有位置信息，例如文件名与行编号，以利寻找与更正拼错的单词。报告将以位置排序，且当它们在同一位置发现多个异常时，则进一步依异常单词排序。
- 应支持用户可指定的后缀缩减，让单词列表保持在易于管理的大小。

在本节最后的例12-4中，我们会展示满足上述所有目标且做得更多、更完整的程序。由于程序功能相当多，因此本节接下来会辅以简单文字来详述细节，并将程序分段以便说明。

使用的测试输入文件包含了 spell 手册页前几段的内容，程序执行结果大致如下：

```
$ awk -f spell.awk testfile
deroff
eqn
ier
nx
tbl
thier
```

或是指定冗长模式，则为：

```
$ awk -f spell.awk -- -verbose testfile
testfile:7:eqn
testfile:7:tbl
testfile:11:deroff
testfile:12:nx
testfile:19:ier
testfile:19:thier
```

12.4.1 介绍性注释

程序会从详尽的注释文字开始，不过在这里，我们仅展示介绍与语法部分：

```
# 实例简单的拼写检查程序，搭配用户可定义的异常列表。
# 内置字典是由标准的 UNIX 拼写字典列表构建而成。
# 不过可以在命令行上覆盖该设置。
#
#
...
#
# 语法 :
#       awk [-v Dictionaries="sysdict1 sysdict2 ..."] -f spell.awk -- \
```

```
#           [=suffixfile1 =suffixfile2 ...] [+dict1 +dict2 ...] \
#           [-strip] [-verbose] [file(s)]
```

12.4.2 主体

程序的主体只有三行，也就是传统awk程序的：初始化、处理与报告：

```
BEGIN   { initialize() }

        { spell_check_line() }

END     { report_exceptions() }
```

程序文件剩余部分的所有细节将交给字母顺序排列的函数处理，但在本节将以逻辑上的顺序描述它。

12.4.3 initialize()

initialize()函数处理程序的初始化工作。

变量NonWordChars里的正则表达式，是用以删除不想要的字符。与ASCII字母与撇号一起，范围在161到255内的字符都被保留作为单词字符，所以ASCII的文件、任何ISO 8859-*n*字符集及以UTF-8编码的Unicode，所有都能被处理而无须关心字符集。

128到160间的字符会被忽略，因为在这之间的所有字符集，都作为额外的控制字符与一个无中断的空格。这些字符集里有部分具有一些在160以上的非字母字符，但使用它们也增加了我们不想要的字符集依赖性。非字母字符是很少见的，即使有，在我们的程序下，最坏的情况就是在拼写异常报告里偶尔会出现错误。

我们假定将进行拼写检查的文件，与其相关联的字典具有相同字符集编码。如果否，则通过iconv，将它们转换为一致的编码。

如果所有的awk实例都遵循POSIX，则我们可以这么设置NonWordChars：

```
NonWordChars = "[^'[:alpha:]]"
```

之后，当前locale会决定应忽略哪些字符。不过这种指定方式不具可移植性，因为有很多awk实现不支持POSIX风格的正则表达式：

在locale导入UNIX前，我们还是能够以否定单词字符集的方式，赋值给NonWordChars：

```
NonWordChars = "[^'A-Za-z\241-\377]"
```

然而，在locale的出现下，正则表达式里字符范围是按照locale类型而被解释，所以其值在各平台间的结果可能不一致。解决方式是改用明白地列举字符的方式，把赋值编写

为连续的字符串，并适当地对齐，以利于人工迅速识别否定字符集里的字符。我们使用八进制表示 127 以上的值，因为这么做会比混杂重音字符要来得清楚许多。

initialize()接下来会识别并载入字典，并处理命令行参数与后缀规则。

```
function initialize()
{
   NonWordChars = "[^" \
     "'" \
     "ABCDEFGHIJKLMNOPQRSTUVWXYZ" \
     "abcdefghijklmnopqrstuvwxyz" \
         "\241\242\243\244\245\246\247\250\251\252\253\254\255\256\257" \
     "\260\261\262\263\264\265\266\267\270\271\272\273\274\275\276\277" \
     "\300\301\302\303\304\305\306\307\310\311\312\313\314\315\316\317" \
     "\320\321\322\323\324\325\326\327\330\331\332\333\334\335\336\337" \
     "\340\341\342\343\344\345\346\347\350\351\352\353\354\355\356\357" \
     "\360\361\362\363\364\365\366\367\370\371\372\373\374\375\376\377" \
     "]"
    get_dictionaries()
    scan_options()
    load_dictionaries()
    load_suffixes()
    order_suffixes()
}
```

12.4.4 get_dictionaries()

get_dictionaries()会填入默认系统字典的列表：我们提供的是两个方便取得的文件。用户可以通过提供字典列表作为命令行变量 Dictionaries 的值，或直接使用 DICTIONARIES 环境变量，而使得该默认选择失效。

如果 Dictionaries 为空，我们会查阅环境数组 ENVIRON，并使用其内所设置的值。如果这么做 Dictionaries 依然为空，我们就会提供一个内置列表。该列表的选择必须花点心思，因为在不同的 UNIX 平台间会出现极大差异，而且在文件很小时，此程序所消耗的大部分执行期时间是在载入字典。除此之外，Dictionaries 应包含一个以空白分隔的字典文件名列表，我们会将其切割，并存储在全局性 DictionaryFiles 数组里。这里选择的单词列表是在我们某些系统里 spell 所使用的单词列表（约 25 000 条记录），以及 Donald Knuth 提供的一个较大型列表（约 110 000 条记录，注 6）。

请留意字典名称是如何被存储的：它们是数组索引（indices），而非数组值（value）。这么设计的理由有二：第一，它可以自动处理提供字典超过一次以上的情况，只有文件名的一个实体被存储；第二，它可易于使用 for (*key* in *array*)循环，迭代经过整个字典列表。无须维护用于计算字典数目的变量。

注 6：　可从 *ftp://labrea.stanford.edu/pub/dict/words.gz* 取得。

这是代码：

```
function get_dictionaries(        files, key)
{
    if ((Dictionaries == "") && ("DICTIONARIES" in ENVIRON))
        Dictionaries = ENVIRON["DICTIONARIES"]
    if (Dictionaries == "")      # 使用默认的字典列表
    {
        DictionaryFiles["/usr/dict/words"]++
        DictionaryFiles["/usr/local/share/dict/words.knuth"]++
    }
    else                 # 使用命令行提供的系统字典
    {
        split(Dictionaries, files)
        for (key in files)
            DictionaryFiles[files[key]]++
    }
}
```

12.4.5 scan_options()

scan_options()处理的是命令行。该函数预期会找到选项(-strip与/或-verbose)、用户字典（以UNIX spell传统的开头+指定)、后缀规则文件（前置=标记）及要作拼写检查的文件。任何的-v选项所设置的Dictionaries变量都已由awk处理，且不在参数数组ARGV中。

scan_options()里的最后一个语句得解释一下：在测试期间，我们发现如果ARGV结尾处留有空参数时，nawk不会读取标准输入，但gawk与mawk会。因此我们在ARGC上减一，直到ARGV的结尾有一个非空的参数：

```
function scan_options(         k)
{
    for (k = 1; k < ARGC; k++)
    {
        if (ARGV[k] = = "-strip")
        {
            ARGV[k] = ""
            Strip = 1
        }
        else if (ARGV[k] = = "-verbose")
        {
            ARGV[k] = ""
            Verbose = 1
        }
        else if (ARGV[k] ~ /^=/)        # 后缀文件
        {
            NSuffixFiles++
            SuffixFiles[substr(ARGV[k], 2)]++
            ARGV[k] = ""
```

```
        }
        else if (ARGV[k] ~ /^[+]/)        # 私人字典
        {
            DictionaryFiles[substr(ARGV[k], 2)]++
            ARGV[k] = ""
        }
    }

    # 删除结尾的空参数 (针对 nawk)
    while ((ARGC > 0) && (ARGV[ARGC-1] = = ""))
        ARGC--
}
```

12.4.6 load_dictionaries()

load_dictionaries()会从所有的字典中读取单词列表。此代码相当简单：外部循环处理DictionaryFiles数组，内部循环则利用getline一次读取一行。每一行正好包含已确认拼写正确的一个单词。字典只建立一次，之后重复使用，所以我们假定行内不会有空白字符，所以无须试图删除它。每个单词都被转换为小写，且存储为全局性Dictionary数组的一个索引。无须另外计数数组中的记录，因为数组只被用在成员测试中的其他地方。以这类测试而言，所有各种程序语言中所提供的数据结构中，关联数组是最快，且最利落的处理方式：

```
function load_dictionaries(        file, word)
{
    for (file in DictionaryFiles)
    {
        while ((getline word < file) > 0)
            Dictionary[tolower(word)]++
        close(file)
    }
}
```

12.4.7 load_suffixes()

很多语言的单词都可以通过切开后缀，而被简化为更短的根单词 (root words)。例如在英文里，*jumped*、*jumper*、*jumpers*、*jumpier*、*jumpiness*、*jumping*、*jumps* 及 *jumpy* 的根单词都为*jump*。后缀有时也会改变单词的最终字母：*try*为*triable*、*trial*、*tried*与*trying*的根。如此，我们需要存储在字典里的基础单词集，就会比包含后缀的单词集小好几倍。由于I/O与计算机计算比起来相对较慢，因此我们认为，在程序里处理后缀以缩短字典大小，及减少异常列表里错误报告的数目，是值得付出的。

load_suffixes()处理后缀规则的载入。不同于载入字典的是：我们在这里可能提供内置的规则，而非从文件中读取。因此，我们保留数组里的记录数目的全局性计数，该数组是保存后缀规则的文件名。

后缀规则会带有解释，且为了说明，我们呈现的是传统的英文规则，见例12-3。我们以正则表达式匹配后缀，每一个单词末端都带有$锚点（anchor）。当后缀被截断时，则必须提供一个替换后缀，例如把*tr+ied*缩短成*tr+y*，且时常会有数种可能的替换。

例 12-3：英文的后缀规则：english.sfx

```
'$                              # Jones' -> Jones
's$                             # it's -> it
ably$           able            # affably -> affable
ed$             "" e            # breaded -> bread, flamed -> flame
edly$           ed              # ashamedly -> ashamed
es$             "" e            # arches -> arch, blues -> blue
gged$           g               # debugged -> debug
ied$            ie y            # died -> die, cried -> cry
ies$            ie ies y        # series -> series, ties -> tie, flies -> fly
ily$            y ily           # tidily -> tidy, wily -> wily
ing$                            # jumping -> jump
ingly$          "" ing          # alarmingly -> alarming or alarm
lled$           l               # annulled -> annul
ly$             ""              # acutely -> acute
nnily$          n               # funnily -> fun
pped$           p               # handicapped -> handicap
pping$          p               # dropping -> drop
rred$           r               # deferred -> defer
s$                              # cats -> cat
tted$           t               # committed -> commit
```

因此，后缀规则的最简单规格是以正则表达式进行后缀匹配，紧接着一个以空白字符分隔的替换列表。因为可能的替换之一是空字符串，我们以`""`表示。如果它是唯一的替换，那我们会省略它。英文是高度不规则且拥有大量外来语的语言，所以有许多后缀规则，且绝对比我们在`english.sfx`里列的还多很多。不过后缀列表仅用于降低错误报告的发生，因为它有效地延展字典大小、不会影响程序的正确运行。

为了便于人类管理后缀规则文件，其规则必须可使用注释扩展以提供它们的应用范例。我们遵循一般UNIX的注释实例，也就是从`#`到行结尾都为注释。因此，`load_suffixes()`会截去注释与开头、结尾的空白字符，再丢弃空白行。留下来的是正则表达式与零到多个替换的列表，用于其他地方可以调用awk内置的字符串替换函数`sub()`。该替换列表是以空白分隔的字符串被存储，可供我们稍候在其上应用`split()`内置函数。

后缀替换可使用`&`表示匹配文字，不过我们在`english.sfx`中并未举出此功能的例子。

我们曾考虑让`load_suffixes()`提供正则表达式里的`$`锚点，但终究推翻这个想法，因为这么可能会限制其他语言所要求的后缀匹配的规格。后缀规则文件必需时时刻刻投入相当的关注，但这个工作只需要在每种语言里做一次就好。

遇到未提供后缀文件的情况时，我们会载入默认的后缀集，具有空的替换值。split()内置函数有助于缩短该初始化的代码：

```
function load_suffixes(        file, k, line, n, parts)
{
    if (NSuffixFiles > 0)       # 自文件中载入后缀正则表达式
    {
        for (file in SuffixFiles)
        {
            while ((getline line < file) > 0)
            {
                sub(" *#.*$", "", line)   # 截去注释
                sub("^[ \t]+", "", line)  # 截去开头空白
                sub("[ \t]+$", "", line)  # 截去结尾空白
                if (line =  = "")
                    continue
                n = split(line, parts)
                Suffixes[parts[1]]++
                Replacement[parts[1]] = parts[2]
                for (k = 3; k <= n; k++)
                    Replacement[parts[1]] = Replacement[parts[1]] " " \
                        parts[k]
            }
            close(file)
        }
    }
    else                        # 载入英文后缀正则表达式的默认表格
    {
        split("'$ 's$ ed$ edly$ es$ ing$ ingly$ ly$ s$", parts)
        for (k in parts)
        {
            Suffixes[parts[k]] = 1
            Replacement[parts[k]] = ""
        }
    }
}
```

12.4.8 order_suffixes()

后缀替换得小心地处理：特别是它应该以“自第一个匹配到最长（longest-match-first）”的算法执行。order_suffixes()采取存储在全局性 Suffixes 数组中的后缀规则列表，并复制一份到 OrderedSuffix 数组，以自 1 起始至 NOrderedSuffix 的整数作为数组索引。

然后，order_suffixes()会使用简单的冒泡排序，通过递减模式长度，使用最内部循环里的 swap()函数重新排列 OrderedSuffix 里的记录。swap()的运行很简单：在其参数数组中，交换元素 i 与 j。该排序技巧的复杂度与被排序元素的数量成平方比，不过 NOrderedSuffix 预期不会那么大，所以该排序不可能耗费如此显著的程序执行期：

```
function order_suffixes(         i, j, key)
{
    # 以渐减的长度排列后缀
    NOrderedSuffix = 0
    for (key in Suffixes)
        OrderedSuffix[++NOrderedSuffix] = key
    for (i = 1; i < NOrderedSuffix; i++)
        for (j = i + 1; j <= NOrderedSuffix; j++)
            if (length(OrderedSuffix[i]) < length(OrderedSuffix[j]))
                swap(OrderedSuffix, i, j)
}

function swap(a, i, j,         temp)
{
    temp = a[i]
    a[i] = a[j]
    a[j] = temp
}
```

12.4.9 spell_check_line()

我们已介绍完程序必备的起始代码。在程序启动时的第二组模式/操作会调用spell_check_line()处理输入数据流的每一行。

第一个工作便是将此行减为一个单词列表。内置函数gsub()只需在一行代码中将非文数字的字符删除，即可完成此任务。产生的单词可以通过$1，$2，…，$NF取用，因此只要一个简单的for循环即可重复处理这些单词，将它们交给spell_check_word()个别处理。

以一般awk程序惯例来说，我们避免在函数体里引用匿名式的数字型字段名称，例如$1，而倾向于将它们限制在较短的操作代码块里。不过有个异常：$k，它是整个程序里唯一这类匿名式引用。为了避免修改时发生的不必要记录重组，我们复制一份到本地变量后，随即截去外部的撇号字符（缩写的一撇），并传送任何非空的结果给spell_check_word()进行下一步处理：

```
function spell_check_line(         k, word)
{
    gsub(NonWordChars, " ")                  # 消除非单词字符
    for (k = 1; k <= NF; k++)
    {
        word = $k
        sub("^'+", "", word)                 # 截去开头的撇号字符
        sub("'+$", "", word)                 # 截去结尾的撇号字符
        if (word != "")
            spell_check_word(word)
    }
}
```

一旦单词已被认可，则字符特有的特殊处理方式并不见得好。但撇号字符在某些语言里指的只是省略，以及外部引号，其实它是个过载的字符。使用消除它的引文，可以降低最后拼写异常列表里错误报告的数量。

截去撇号字符对荷语来说不可行，它有小部分单词是将撇号置于单词初始位置：`*n*对于*een*、*s*对于*des*，以及`*t*对于*het*。这些都是些琐碎细节，你可以通过增加异常字典的方式处理。

12.4.10 spell_check_word()

spell_check_word()是真正进行处理的地方，不过在大部分情况下，这部分处理很快。如果在全局性Dictionary数组里发现小写单词且拼写正确，则我们可立即返回。

如果该单词未出现于单词列表中，即可能是拼写异常。但如果用户要求截去后缀，那么我们就得再做点什么。strip_suffixes()函数的功能，即用以产生一个或多个相关单词列表，并存储为本地wordlist数组的索引。for循环接着会处理这个列表，如果它发现Dictionary数组里有这些单词，则返回。

如果无须截去后缀，或我们未于字典中发现任何替换单词，则该单词确定就是拼写异常了。此时将其写到输出报告并非好的做法，因为我们通常会想要一份排序后没有重复的拼写异常列表。例如*awk*这个单词在本章已出现超过30次，但在所有标准UNIX拼写字典里都找不到它。因此我们将这个单词存储到全局性Exception数组中，当用户要求以冗长模式输出结果时，我们可以在该单词前置一个位置，其由一个冒号终结的文件名与行编号所定义。这种报告的形式在许多UNIX工具里很常见，也易于让人们与聪明的文字编辑程序迅速了解。需留意的一点是：虽然字母大小写在字典查找作业里被忽略，但在此报告中会保留原始字母的大小写：

```
function spell_check_word(word,        key, lc_word, location, w, wordlist)
{
    lc_word = tolower(word)
    if (lc_word in Dictionary)            # 可接受的拼写
        return
    else                                  # 可能的异常
    {
        if (Strip)
        {
            strip_suffixes(lc_word, wordlist)
            for (w in wordlist)
                if (w in Dictionary)
                    return
        }
        location = Verbose ? (FILENAME ":" FNR ":") : ""
        if (lc_word in Exception)
```

```
            Exception[lc_word] = Exception[lc_word] "\n" location word
        else
            Exception[lc_word] = location word
    }
}
```

12.4.11 strip_suffixes()

发现不在字典里的单词，且指定-strip选项时，我们会调用strip_suffixes()应用后缀规则。其循环是以递减的后缀长度，依次处理后缀正则表达式。如果单词匹配，则后缀会被删除，以便取得根单词。如果无替换的后缀，则单词将存储为wordlist数组的索引。否则，我们会将替换列表切分为各个数组成员，并依次附加每个替换到根单词中，将它增加到wordlist数组。我们还需在内部循环里处理一个特殊情况，检查是否有特殊的双字符（two-character）字符串""，这里我们会以空字符串取代。一旦匹配成功，break语句即离开循环，且函数会将其返回给调用者。否则，循环会继续下一个后缀正则表达式。

我们可以让这个函数针对wordlist里存储的单词进行字典查找。我们不这么做是因为这样会将查找与后缀处理混在一起，且程序将很难扩展以显示替换的侯选单词（UNIX的spell提供-x选项完成此工作：处理每一个可采用后缀的输入单词，它会产生具有相同根的正确拼音单词列表）。

虽然后缀规则足以应付许多印欧语系，但其他语言则完全不需要，更有另外一些语言在单词拼法上需作更复杂的改动，不管是在文法的格、数词或是时态上。对这类语言，最简单的方式似乎就是提供更充分的字典了。

此处为代码：

```
function strip_suffixes(word, wordlist,        ending, k, n, regexp)
{
    split("", wordlist)
    for (k = 1; k <= NOrderedSuffix; k++)
    {
        regexp = OrderedSuffix[k]
        if (match(word, regexp))
        {
            word = substr(word, 1, RSTART - 1)
            if (Replacement[regexp] == "")
                wordlist[word] = 1
            else
            {
                split(Replacement[regexp], ending)
                for (n in ending)
                {
                    if (ending[n] == "\"\"")
```

```
                        ending[n] = ""
                    wordlist[word ending[n]] = 1
                }
            }
            break
        }
    }
}
```

12.4.12 report_exceptions()

程序最后的任务，会从最后三组模式/操作开始。report_exceptions()建立带有命令行选项的sort 管道处理，视用户是否要求具有唯一性异常单词的完整列表，或报告是否需要带有位置信息的冗长模式。无论是哪一种情况，我们提供了-f选项予sort，令其忽略大小写，还有-u选项，取得具唯一性（不重复）的输出行。简单的for循环将异常列表输出至管道，最后的close()会关闭管道且完成程序。

这是代码：

```
function report_exceptions(         key, sortpipe)
{
    sortpipe = Verbose ? "sort -f -t: -u -k1,1 -k2n,2 -k3" : \
                "sort -f -u -k1"
    for (key in Exception)
        print Exception[key] | sortpipe
    close(sortpipe)
}
```

例 12-4 包含了完整的代码，完成我们的拼写检查程序。

例 12-4：拼写检查程序

```
# 实例简单的拼写检查程序，搭配用户可指定的异常列表。
# 内置字典是以标准的 UNIX 拼写字典列表构建。
# 不过可以在命令行上覆盖它。
#
...
# 语法 ：
#       awk [-v Dictionaries="sysdict1 sysdict2 ..."] -f spell.awk -- \
#           [=suffixfile1 =suffixfile2 ...] [+dict1 +dict2 ...] \
#           [-strip] [-verbose] [file(s)]

BEGIN   { initialize() }

        { spell_check_line() }

END     { report_exceptions() }
function get_dictionaries(          files, key)
{
    if ((Dictionaries == "") && ("DICTIONARIES" in ENVIRON))
```

```
        Dictionaries = ENVIRON["DICTIONARIES"]
    if (Dictionaries == "")       # 使用默认目录列表
    {
        DictionaryFiles["/usr/dict/words"]++
        DictionaryFiles["/usr/local/share/dict/words.knuth"]++
    }
    else                     # 使用来自命令行的系统目录
    {
        split(Dictionaries, files)
        for (key in files)
            DictionaryFiles[files[key]]++
    }
}

function initialize()
{
   NonWordChars = "[^" \
     "'" \
     "ABCDEFGHIJKLMNOPQRSTUVWXYZ" \
     "abcdefghijklmnopqrstuvwxyz" \
         "\241\242\243\244\245\246\247\250\251\252\253\254\255\256\257" \
     "\260\261\262\263\264\265\266\267\270\271\272\273\274\275\276\277" \
     "\300\301\302\303\304\305\306\307\310\311\312\313\314\315\316\317" \
     "\320\321\322\323\324\325\326\327\330\331\332\333\334\335\336\337" \
     "\340\341\342\343\344\345\346\347\350\351\352\353\354\355\356\357" \
     "\360\361\362\363\364\365\366\367\370\371\372\373\374\375\376\377" \
     "]"
    get_dictionaries()
    scan_options()
    load_dictionaries()
    load_suffixes()
    order_suffixes()
}

function load_dictionaries(        file, word)
{
    for (file in DictionaryFiles)
    {
        while ((getline word < file) > 0)
            Dictionary[tolower(word)]++
        close(file)
    }
}

function load_suffixes(        file, k, line, n, parts)
{
    if (NSuffixFiles > 0)       # 自文件载入后缀正则表达式
    {
        for (file in SuffixFiles)
        {
            while ((getline line < file) > 0)
            {
                sub(" *#.*$", "", line)    # 截去注释
                sub("^[ \t]+", "", line)  # 截去前置空白字符
```

```
                sub("[ \t]+$", "", line)  # 截去结尾空白字符
                if (line =  = "")
                    continue
                n = split(line, parts)
                Suffixes[parts[1]]++
                Replacement[parts[1]] = parts[2]
                for (k = 3; k <= n; k++)
                    Replacement[parts[1]] = Replacement[parts[1]] " " \
                        parts[k]
            }
            close(file)
        }
    }
    else                  # 载入英文后缀正则表达式的默认表格
    {
        split("'$ 's$ ed$ edly$ es$ ing$ ingly$ ly$ s$", parts)
        for (k in parts)
        {
            Suffixes[parts[k]] = 1
            Replacement[parts[k]] = ""
        }
    }
}

function order_suffixes(        i, j, key)
{
    # 以递减的长度排列后缀
    NOrderedSuffix = 0
    for (key in Suffixes)
        OrderedSuffix[++NOrderedSuffix] = key
    for (i = 1; i < NOrderedSuffix; i++)
        for (j = i + 1; j <= NOrderedSuffix; j++)
            if (length(OrderedSuffix[i]) < length(OrderedSuffix[j]))
                swap(OrderedSuffix, i, j)
}

function report_exceptions(        key, sortpipe)
{
    sortpipe = Verbose ? "sort -f -t: -u -k1,1 -k2n,2 -k3" : \
                "sort -f -u -k1"
    for (key in Exception)
        print Exception[key] | sortpipe
    close(sortpipe)
}

function scan_options(        k)
{
    for (k = 1; k < ARGC; k++)
    {
        if (ARGV[k] == "-strip")
        {
            ARGV[k] = ""
            Strip = 1
        }
```

```
        else if (ARGV[k] == "-verbose")
        {
            ARGV[k] = ""
            Verbose = 1
        }
        else if (ARGV[k] ~ /^=/)          # 后缀文件
        {
            NSuffixFiles++
            SuffixFiles[substr(ARGV[k], 2)]++
            ARGV[k] = ""
        }
        else if (ARGV[k] ~ /^[+]/)        # 私有字典
        {
            DictionaryFiles[substr(ARGV[k], 2)]++
            ARGV[k] = ""
        }
    }

    # 删除结尾的空参数 (for nawk)
    while ((ARGC > 0) && (ARGV[ARGC-1] == ""))
        ARGC--
}

function spell_check_line(        k, word)
{
    gsub(NonWordChars, " ")                 # 消除非单词字符
    for (k = 1; k <= NF; k++)
    {
        word = $k
        sub("^'+", "", word)                # 截去前置的撇号字符
        sub("'+$", "", word)                # 截去结尾的撇号字符
        if (word != "")
            spell_check_word(word)
    }
}

function spell_check_word(word,        key, lc_word, location, w, wordlist)
{
    lc_word = tolower(word)
    if (lc_word in Dictionary)              # 可接受的拼写
        return
    else                                    # 可能的异常
    {
        if (Strip)
        {
            strip_suffixes(lc_word, wordlist)
            for (w in wordlist)
                if (w in Dictionary)
                    return
        }
        location = Verbose ? (FILENAME ":" FNR ":") : ""
        if (lc_word in Exception)
            Exception[lc_word] = Exception[lc_word] "\n" location word
        else
```

```
            Exception[lc_word] = location word
    }
}

function strip_suffixes(word, wordlist,        ending, k, n, regexp)
{
    split("", wordlist)
    for (k = 1; k <= NOrderedSuffix; k++)
    {
        regexp = OrderedSuffix[k]
        if (match(word, regexp))
        {
            word = substr(word, 1, RSTART - 1)
            if (Replacement[regexp] == "")
                wordlist[word] = 1
            else
            {
                split(Replacement[regexp], ending)
                for (n in ending)
                {
                    if (ending[n] == "\"\"")
                        ending[n] = ""
                    wordlist[word ending[n]] = 1
                }
            }
            break
        }
    }
}

function swap(a, i, j,        temp)
{
    temp = a[i]
    a[i] = a[j]
    a[j] = temp
}
```

12.4.13 回顾拼写检查程序

UNIX拼写检查程序第一版所使用的就是我们在本章一开始所呈现的管道处理。在文件 *The UNIX Heritage Society*（注 7）中，可以找到第一版以 C 写成的 UNIX 拼写程序，此即为 1975 Version 6 UNIX 的 typo 命令，约为 350 行的 C 代码。spell 首次出现是在 1979 Version 7 UNIX 版本，大约 700 行的 C 代码。spell 在 1995 4.4 BSD-Lite 的源代码版本中遭删除，推测可能是因为商业机密或者版权上的问题。

现代的OpenBSD spell约有1100行C代码，在它的三个基本字典中各有30多个单词。

注 7： 见 *http://www.tuhs.org/*。

GNU ispell版本3.2约为13500行的C代码，及GNU的aspell版本0.60则约为29500行的C++与C代码。这两个程序都已国际化，都附有10至40种语言的字典。ispell拥有相当大的英文字典，约80000个一般单词，搭配3750个左右美语与英语的变化体。aspell字典就更大了：142000个英文单词，辅以4200个来自美国、英国与加拿大语系的变体。

我们的拼写检查程序spell.awk是一个真正卓越的程序，你会觉得幸好有它，如果你重新以其他程序语言编写这样的程序，就会了解awk真的比较好。就如同Johnson在1975年的原始spell命令，我们的设计与实例不到一个下午就完成。

约190行代码，搭配三组模式/操作的单命令行程序与11个函数，它能做到传统UNIX的spell能做的所有事，甚至更多：

- 具有-verbose选项，程序会报告拼写异常的位置信息。
- 用户可控制字典，让程序可立即应用于复杂的技术文件及以英文以外的语言所写成的文章上。
- 用户可定义后缀列表，有助于拼写检查国际化，以及提供用户控制后缀缩减，就像所有平台都提供的一些拼写检查程序一样。
- 所有相关联的字典与后缀文件都为简单的文本文件，可使用任何文本编辑器修改，且大部分UNIX文本工具都能处理。部分拼写检查程序仍保留二进制形式的字典文件，这会使单词列表难以检阅、维护与更新，也很难再用作其他目的。
- 主要依赖字符集的是，这是低于128以下的ASCII顺序的初始化假设。尽管不再支持IBM大型主机EBCDIC，European 8-bit字符集也不会带来什么问题，甚至以多字节UTF-8编码的两百万字符Unicode字集也可适当处理它，但要确切认知与删除非ASCII Unicode的标点符号仍需多费心思。由于多字节字符集的复杂性，且可能在任何地方都需要用到，这部分的功能应另以独立工具实现，作为使用spell.awk前的预先过滤程序。
- 输出排序的顺序，对某些语言来说是个复杂议题，这个操作完全由sort命令决定，而该命令又受当前环境的locale设置所影响。最好的情况就是：某个独立工具本地化处理排序的复杂问题，这么一来其他软件，包括我们的程序，便无须理会这个争议了。这也就是我们在1.2节里所说的："让别人去做困难的部分"。
- 虽然是以解释式语言编写，我们的程序算是相当快的了。在2 GHz Pentium 4的工作站上，使用mawk，它只需要一秒便能检查本书所有文件的拼写，时间仅为OpenBSD的spell的1.3倍及GNU ipsell的2.0倍。

  以执行上的概况来看（见12.4.14节），载入字典需花费总时间的5%，而每15个单

词就有一个在字典上找不到。加入 -strip选项会增加约25%的执行期及减少相同量的输出大小。每70个单词里有一个通过strip_suffixes()里的match()测试。

- 后缀支持在这190行的代码里约占去90，所以我们可以说，写这个方便好用的多语言拼写检查程序大约只用了100行的 awk 代码。

上述特性列表以及我们的程序，最明显缺乏的功能就是截去文件标记（markup），有些拼写检查提供此功能。我们故意不这么做，是因为它完全违背了UNIX的传统：一个（小）工具只做一件事。标记删除其实是很有用的，所以值得另写一个独立的过滤程序，例如dehtml、deroff、desgml、detex以及dexml。这之中，只有deroff在大部分UNIX系统下找得到，不过其他的实现也只要几行 awk 就办得到。

撇开三个简单的 substr()调用不谈，我们的程序另缺的就是独立字符的处理。在C里这类处理的需求，很多其他语言也是，正是主要的bug来源。

该程序只剩这些事待完成：累积适当数量的字典集与其他语言的后缀列表，通过Shell脚本包装（wrapper）的提供，让它的用户界面看起来更像传统的UNIX程序，还有编写手册页。虽然我们在这里没有将它们呈现出来，不过本书范例程序已提供包装程序与手册页。

12.4.14 awk 程序的性能

我们以几个与awk程序性能有关的评论作总结。awk程序就像其他脚本语言，先被编译为简洁的内部表示，再将该表示在执行期时，通过小型虚拟机器（virtual machine）解释。内置函数是以底层实现语言写成，现行的C为通用版本，执行速度等同于本地软件的速度。

程序的性能不单单指计算机上的时间，人们花在上头的时间也算。如果是花一个小时，以awk写一个只要执行几秒的程序，相对于以编译语言花数个小时编写、调试所写成的相同程序，结果只是少几秒的执行期，那么人们花在上头的时间就是性能的重点考虑了。对许多软件工具而言，awk赢在它有大量的手段与方法足以完成任务。

传统像是 Fortran 与 C 这类的编译语言，内层代码会与底层机器语言息息相关，程序设计老手们马上感觉得到孰优孰劣。算法与内存操作的数量、循环嵌套设计有多深，都为重点，且易于计算，并与执行期直接相关。以数字方面的程序为例，一般法则是代码的10%，耗费90%的执行期：而这10%的代码就称为热点（hot spot）。像是将最内部循环的常用表达式拉出来，还有重新排列运算以符合存储配置等，这些最佳化的操作，有时可大大促进执行时间。然而，高级语言、使用大量函数调用的语言（例如Lisp，它的每条语句都是函数）或解释式语言，它们的执行期都较难以估算，也很难识别出它们的热点。

做许多模式匹配的awk程序，通常也被该运算的复杂度所限制，其完全是以原始的速度执行。这类程序很少能够通过编译程序，像C或C++的重写，再做点什么改善。我们所提及的三种awk实例，都各自独立编写而成，且对于特定语句，也有完全不同的执行时间。

由于我们使用awk写了很多软件工具，有部分用以处理动辄以GB计的数据，执行期性能对我们来说有时是相当重要的。几年前，我们之中有人（NHFB）想做的是pawk（注8），它其实是最小型的实例，nawk的探测版。pawk会报告语句计数与时间。我们其他人（AR），也自动自发将类似的语句计数支持加入到GNU的gawk，因此，pgawk已自3.1.0版开始成为标准配备。pgawk会产生输出探测（profile）至awkprof.out，再搭配带有语句执行计数评注的程序列表。计数很快便会识别出热点。计数为零（或空）即表示代码完全未执行，所以输出探测还可以当作测试涵盖范围（test coverage）的报告。当测试文件是用以验证所有程序语句是否在检测期间都被执行时，这样的报告就很重要了：潜藏于代码中的bug可能很少或甚至从未执行过。

精准的执行时间是很难取得的，因为典型的CPU计时器（timer）仅能得到每秒60到100次的核对，这在GHz处理器的时代完全不够用。幸好，已有部分UNIX系统提供低成本的十亿分之一秒解析计时器，而pawk在这些平台上都使用它们。

12.5 小结

原始的拼写检查雏型，展现了UNIX软件工具应用的优雅与能力。只花一个下午的时间，就能做出这样一个方便又有用的单一目的程序。在体验过以Shell写成的雏型后，再以C重写一份在线版本也是常有的事。

私有字典的使用，是UNIX spell里一个强而有力的功能。虽然UNIX环境后台下的locale设置，很可能导出奇怪的行为模式，但字典的使用仍有其价值，事实上本书的各个章节，我们都建立了私有字典，以便管理拼写检查的工作。

可自由取用的ispell与aspell程序相当大，功能也很强，但却缺乏一些让批处理模式更好用的功能。我们展示过如何封装简单的Shell脚本，所以我们可以解决这样的不足，让程序更适于我们的需求。此是Shell脚本的最传统用法：取一个几乎能完成所有所需工作的程序，然后稍微修改它的结果，以完成剩下的工作。这也符合我们在软件工具设计原则里所说的："让别人完成困难的部分"。

最后，awk拼写检查程序完美展现了该语言的优雅与强大功能。就在某个午后，NHFB完成了低于200行，可以（已经是）正式使用于拼写检查的程序。

注8：　可在 *http://www.math.utah.edu/pub/pawk/* 取得。

第13章

进程

进程(process)指的是执行中程序的一个实例(instance)。新进程由fork()与execve()等系统调用所起始,然后执行,直到它们下达exit()系统调用为止。系统调用fork()与execve()的处理相当复杂,且不是本书所要说明的范畴,如果你有兴趣,可以参考它们的网页。

UNIX系统都支持多进程。虽然计算机看来像是一次做了很多事,但除非是它拥有多个CPU,否则这只是错觉。事实上,每个进程仅容许在一个极短的期间执行,我们称为时间片段(time slice),之后进程会先暂时搁置,让其他等待中的进程执行。时间片段极短,通常只有几微秒,所以人们很少感觉得到进程将控制权交回kernel,再交给另一个进程的这种文本切换(context switches)。进程本身不管本文切换这件事,也没有必要在程序里撰写撤回控制权予操作系统的处理。

操作系统内核里,称为调度器(scheduler)的部分负责管理进程的执行。当出现多CPU时,调度器会试着使用所有CPU处理工作负载。用户除了觉得响应速度的改善之外,多半不会查觉有何不同。

进程会被指定优先权,这么一来,有时间考虑的进程便能比不重要的进程先执行。nice与renice命令即用于调整进程的优先权。

在任何瞬间,等待执行之进程的平均数,被称为平均负载(load average),最简单的uptime命令便能显示:

```
$ uptime                                    显示开机至今的时间、用户数,及平均负载
  1:51pm up 298 day(s), 15:42, 32 users, load average: 3.51, 3.50, 3.55
```

由于平均负载会一直变化,uptime会回报三个平均时间估算值,分别为最后一分钟、五分钟及十五分钟的估算值。当平均负载持续地超出可用CPU的承载时,表示系统工作已超出它所能负荷的了,此时响应可能会陷入停滞不前的状态。

讨论操作系统的书，对进程与调度器有较深入的探讨，在本书中讲得太深入会有些文不对题，对大部分用户而言，也没有必要。所以我们在本章，只说明如何建立进程、列出进程，及删除进程。除此之外，还会介绍将信号传递给进程的方式，及如何监控进程的执行。

13.1 进程建立

UNIX 对计算机运算世界最大的贡献，就是能够轻易地建立进程。此举有助于用户为庞大的工作先撰写小型程序处理各个部分，再将它们整合完成整个工作。由于程序设计的复杂度与日俱增，以小型程序的方式处理会更好写、更好除虫，也更易于了解。

很多程序都由 Shell 启动：每个命令行里的第一个单词是识别要执行的程序。一个命令 Shell 所起始每个进程，都会以下列保证事项启动：

- 进程具有一个内核本文（kernel context）：在内核里的数据结构，会记录与进程相关的信息，让内核便于管理与控制进程的执行。
- 进程拥有一个私用的（private）、被保护的（protected）虚拟地址空间，它可能就像机器可定址空间那么大。不过，其他资源的限制，像是实例内存与外部储存设备上的 swap 空间所组合的大小，其他执行中工作的大小，或是系统调校参数的本地端设置，都会加诸进程执行上的限制。
- 三个文件描述代码（标准输入、标准输出，与标准错误输出）都已开启，且立即可用。
- 起始于交谈模式 Shell 的进程，会拥有一个控制终端机（controlling terminal），其扮演三个标准文件数据流的默认来源处与目的地。控制终端机是让用户可将信号传送给进程，这部分主题在稍后 13.3 节里将会介绍。
- 命令行参数里的通配字符会被展开。
- 内存的一个环境变量区域会存在，包含具有键与值（key/value）指定的字符串，可通过程序库调用取得（在 C 里，为 `getenv()`）。

这些保证没有任何差别待遇：所有执行于相同优先权层级的进程都一视同仁，且进程可以由任何程序写成。

私有地址空间（private address space）可确保进程不受其他进程或内核干扰。未提供这样保障的操作系统很容易出错。

这三个已开启的文件，对大部分的程序来说已经够用，可以使用它们而无需烦恼文件开启与关闭的操作，也不需要知道任何文件名语法或文件系统。

由 Shell 展开的通配字符会免除程序的很多负担，也提供了统一性的命令行处理。

环境空间（environment space）是除了命令行与输入文件之外，可提供信息给进程的另一种方式。

13.2 进程列表

列出进程中最重要的命令便是进程状态（process status）命令：ps。长久以来，ps 的形式发展出主要的两种：System V 式与 BSD 式。很多系统两者都提供，有些则是择一选择性地提供。在 Sun Solaris 系统下：

```
$ /bin/ps                                     System V式的进程状态
   PID TTY      TIME CMD
  2659 pts/60   0:00 ps
  5026 pts/60   0:02 ksh
 12369 pts/92   0:02 bash

$ /usr/ucb/ps                                 BSD 式的进程状态
   PID TT       S  TIME COMMAND
  2660 pts/60   O  0:00 /usr/ucb/ps
  5026 pts/60   S  0:01 /bin/ksh
 12369 pts/92   S  0:02 /usr/local/bin/bash
```

未提供命令行选项，它们的输出就会很相似，只是 BSD 式的细节较多。这里的输出，可以限定为那些与引用（调用）者的进程具有相同用户 ID 及相同控制终端的进程。

ps 是与 ls 的文件列出命令很像，也提供相当多的选项，且在 UNIX 各种平台上都各有不同版本。以 ls 而言，-l 选项输出冗长式数据，是常见用法。如果要取得 ps 的冗长输出，则需要其他选项，在 System V 形式下：

```
$ ps -efl                                         System V风格
 F S  UID PID PPID C PRI NI ADDR  SZ WCHAN STIME TTY   TIME CMD
19 T root   0    0 0   0 SY    ?   0        Dec 27 ?   0:00 sched
 8 S root   1    0 0  41 20    ? 106      ? Dec 27 ?   9:53 /etc/init -
19 S root   2    0 0   0 SY    ?   0      ? Dec 27 ?   0:18 pageout
19 S root   3    0 0   0 SY    ?   0      ? Dec 27 ? 2852:26 fsflush
...
```

在 BSD 形式下，则为：

```
$ ps aux                                  BSD 风格
USER    PID %CPU %MEM   SZ  RSS TT       S   START    TIME COMMAND
root      3  0.4  0.0    0    0 ?        S  Dec 27 2852:28 fsflush
smith 13680  0.1  0.2 1664 1320 pts/25 O 15:03:45    0:00 ps aux
jones 25268  0.1  2.02093619376 pts/24 S   Mar 22   29:56 emacs -bg ivory
brown 26519  0.0  0.3 5424 2944 ?        S  Apr 19    2:05 xterm -name thesis
...
```

这两种形式都允许结合选项字母一起执行，而BSD形式还允许去除选项连字号。这两个例子中，为了符合解说页面，我们都适度地去除了过多的空白。

在这两种类型上，部分设计并不恰当，有时信息太多，但显示的空间却太少：进程起始日期的缩写格式各有不同，而最后一栏的命令有时会被截断，且显示的栏位值有时也会全挤在一起。后者令我们在过滤 ps 的输出时变得很困难。

USER 与 UID 栏为进程拥有者：当你发现进程悬在系统上不动时，这会是关键信息。

PID 为进程 ID 值（process ID），此数字是定义进程的唯一值。在 Shell 中，该数字可以 $$ 表示，我们在其他章节中曾用它产生暂时性文件名。进程 ID 的指定自零起始，每遇到新的进程便加值，直至系统停止。在到达最大可表示的整数值时，进程编号会再从零开始，但避开已被其他进程使用的值。传统的单一用户系统可能只有少数几个活动中的进程，但大型的多用户系统可能就拥有数以千计的进程了。

PPID 为父进程 ID（parent process ID）值：指的是建立此进程的那个进程编号。除了第一个进程外，每个进程都有父进程并会拥有零至多个子进程，所以进程的形式为树状结构。进程编号 0 通常被称为 kernel、sched，或 swapper，而且在某些系统上并不会显示在 ps 的输出结果中。进程编号 1 比较特殊：称为 init，可参考 *init*(8)的使用手册。父进程过早消失（die）的进程，会被重新指定其新的父进程为 init。系统在正常关机下，进程的删除是以由大至小的进程 ID 依次执行，直到只剩下 init 为止，当它结束时，系统便终止。

ps输出的顺序不保证有一定的规则，且由于进程的列表是持续地变动的，每次执行时都会看到不同的输出。

由于进程列表是动态的，许多用户会想要持续观察类似ps这样的文字输出更新状态，或以图形呈现。很多工具程序提供显示这类信息，但没有共通可用的标准工具。最通用的应是 top 了，这是现行许多 UNIX 版本里的标准工具（注 1）。我们认为这个工具和 GNU 的 tar 一样重要，因此如果发现新系统不提供此工具，我们就会立即安装一个。在大部分系统上，top 必须详熟内核数据结构的相关知识，且当操作系统升级时，它也必须更新。top（类似 ps）是少数几个需特殊权限才能执行的程序，某些系统上，会以 setuid 为 root 的方式变通行事。

此处为 top 的输出快照（snapshot），这里显示的多处理器计算机服务器还不算太忙碌：

注 1： 可自 *ftp://ftp.groupsys.com/pub/top* 下载。另一个仅 GNU/Linux 系统适用的实例，可参考 *http://procps.sourceforge.net/*。

```
$ top                                                显示前几名消耗资源的进程
load averages:  5.28,  4.74,  4.59                                   15:42:00
322 processes: 295 sleeping, 4 running, 12 zombie, 9 stopped, 2 on cpu
CPU states: 0.0% idle, 95.9% user, 4.1% kernel, 0.0% iowait, 0.0% swap
Memory: 2048M real, 88M free, 1916M swap in use, 8090M swap free

  PID USERNAME THR PRI NICE  SIZE   RES STATE    TIME    CPU COMMAND
 2518 jones      1   0    0  506M  505M run     44:43 33.95% Macaulay2
 1111 owens      1   0   19   21M   21M run     87:19 24.04% ocDom
23813 smith      1   0   19  184M  184M cpu/0  768:57 20.39% mserver
25389 brown      1   1   19   30M   23M run    184:22  1.07% netscape
...
```

默认状态下，top 会在列表顶端显示 CPU 耗用最多的进程，这通常也就是你想看的那个。不过，top还接受键盘输入，控制排序顺序、限定显示你感兴趣者等等，只要在top通信期（session）下输入 type ?，即可得知你的 top 版本提供了哪些用法。

其他一些列出进程，或显示各类系统负载状态的好用命令，我们呈现于表 13-1。

表 13-1：好用的系统负载命令

系统	命令
所有系统	iostat、netstat、nfsstat、sar、uptime、vmstat、w、xcpustate[注]、xload，与 xperfmon
Apple Mac OS X	pstat
BSD	pstat 与 systat
GNU/Linux	procinfo
HP Alpha OSF/1	vmubc
IBM AIX	monitor
SGI IRIX	gr_osview 与 osview
Sun Solaris	mpstat、perfmeter、proctool、prstat、ptree 与 sdtperfmeter

注：可自 *ftp://ftp.cs.toronto.edu/pub/jdd/xcpustate/* 取得。

大部分情况下，Shell在处理下一个命令之前会等待一进程结束。不过只要在命令最后加入 & 字符，而非分号或换行符号，便能将进程放在后台中执行：我们在 8.2 节的 build-all 脚本里使用过此功能。wait 命令可用以等待某个特定进程完成，在不加任何参数的情况下，则为等待所有后台进程的完成。

虽然本书大多略过 Shell 的交谈模式功能不提，但这里还是要告诉你：bg、fg、jobs，以及 wait 都为处理于目前 Shell 下所建立的执行中进程的 Shell 命令。

有4组键盘字符可用以中断前台进程（foreground processes）。这些字符都可通过stty命令选项而设置，通常为Ctrl-C（intr：杀除）、Ctrl-Y（dsusp：暂时搁置，直到输入更新为止）、Ctrl-Z（susp：暂时搁置），与Ctrl-\（quit：以核心转储（core dump）方式杀除）。

例 13-1 呈现的是简易 top 实例的命令。/bin/sh - 选项所提出的安全性议题，及 IFS 的设置（为换行字符－空格字符－定位字符）与 PATH 指定，已在 8.1 节作过说明。我们需要 BSD 式的 ps，因为它提供的 %CPU 栏，可决定显示顺序，因此设置 PATH 先寻找该版本。PATH 设置在我们所有的系统下几乎都能运行，只有一个例外（SGI IRIX 缺乏 BSD 式的 ps 命令）。

例 13-1：简化的 top 版本

```
#! /bin/sh -
# 持续执行 ps 命令，
# 每次显示之间，只作短时间的暂停
#
# 语法 ：
#       simple-top

IFS='
        '

# 自订 PATH，以先取得 BSD 式的 ps
PATH=/usr/ucb:/usr/bin:/bin
export PATH

HEADFLAGS="-n 20"
PSFLAGS=aux
SLEEPFLAGS=5
SORTFLAGS='-k3nr -k1,1 -k2n'

HEADER="`ps $PSFLAGS | head -n 1`"

while true
do
    clear
    uptime
    echo "$HEADER"
    ps $PSFLAGS |
        sed -e 1d |
            sort $SORTFLAGS |
                head $HEADFLAGS
    sleep $SLEEPFLAGS
done
```

我们将命令选项储存在 HEADFLAGS、PSFLAGS、SLEEPFLAGS 与 SORTFLAGS，以方便某些特定站台客户化。

simple-top输出的解释性标头相当有用，但由于它在ps实例之间有些差异，所以我们

不在脚本里把它写死。取而代之的是：我们只调用 ps 一次，然后把它储存在 HEADER 变量里。

程序其余的部分就是无穷循环了，可使用我们稍早提过的键盘字符中断它。在每个循环重复起始处的 clear 命令会使用 TERM 环境变量的设置，以决定它要传送至标准输出，清除屏幕画面的转义符，将游标留在左上角。uptime 回报平均负载，echo 则提供每栏标头。管道过滤 ps 的输出使用 sed 删除标头行，再依次以 CPU 使用量、username，与进程 ID 排序最后的输出结果，然后仅显示前 20 行。循环里最后的 sleep 命令会产生极短的延迟，不过这和循环重复操作比起来仍是较长的，这么一来此脚本对系统负载的影响才能达到最小。

有时，你会想知道谁在使用你的系统，有多少与哪些进程正执行，而不要 ps 冗长输出所提供的其他额外细节。例 13-2 里的 puser 脚本即可用以产生此类报告：

```
$ puser                                显示用户，及属于他们的进程
albert                 3               -tcsh
                       3               /etc/sshd
                       2               /bin/sh
                       1               /bin/ps
                       1               /usr/bin/ssh
                       1               xload
daemon                 1               /usr/lib/nfs/statd
root                   4               /etc/sshd
                       3               /usr/lib/ssh/sshd
                       3               /usr/sadm/lib/smc/bin/smcboot
                       2               /usr/lib/saf/ttymon
                       1               /etc/init
                       1               /usr/lib/autofs/automountd
                       1               /usr/lib/dmi/dmispd
                       ...
victoria               4               bash
                       2               /usr/bin/ssh
                       2               xterm
```

报告以 username 排序，为了不致于太过混乱，并提升可读性，只有在 username 有变动时才显示 username 值。

例 13-2：puser 脚本

```
#! /bin/sh -
# 显示用户，及其活动中的进程数与进程名称，
# 可选择性地限制显示某些特定用户 (egrep(1)
# username 样式)。
#
# 语法 ：
#       puser [ user1 user2 ... ]

IFS='
'
```

```
PATH=/usr/local/bin:/usr/bin:/bin
export PATH

EGREPFLAGS=
while test $# -gt 0
do
    if test -z "$EGREPFLAGS"
    then
        EGREPFLAGS="$1"
    else
        EGREPFLAGS="$EGREPFLAGS|$1"
    fi
    shift
done

if test -z "$EGREPFLAGS"
then
    EGREPFLAGS="."
else
    EGREPFLAGS="^ *($EGREPFLAGS) "
fi

case "`uname -s`" in
*BSD | Darwin)       PSFLAGS="-a -e -o user,ucomm -x" ;;
*)                   PSFLAGS="-e -o user,comm" ;;
esac

ps $PSFLAGS |
  sed -e 1d |
    EGREP_OPTIONS= egrep "$EGREPFLAGS" |
      sort -b -k1,1 -k2,2 |
        uniq -c |
          sort -b -k2,2 -k1nr,1 -k3,3 |
            awk '{
                  user = (LAST =  = $2) ? " " : $2
                  LAST = $2
                  printf("%-15s\t%2d\t%s\n", user, $1, $3)
                 }'
```

在一贯熟悉的前置处理后，puser脚本开始利用循环搜集可选用的命令行参数，将之传至EGREPFLAGS变量，使用垂直线分隔符表示交替至egrep。循环内容里的if语句处理初始为空字符串的情况，以避免产生空的egrep样式。

参数搜集循环完成后，我们会检查EGREPFLAGS：如果它为空，则重新指定它为符合任何的样式。否则，改为只对行的起始处进行样式比对，并要求结尾加上空格字符，以避免因username开头字符相同而出现错误符合，例如jon与jones。

case语句是在处理ps选项的实例差异。我们希望输出形式为只显示两个值：username与命令名称。BSD系统与BSD扩展的Mac OS X（Darwin）系统要求的选项与其他所有我们测试过的系统有些许不同。

以下7阶段的管道处理报告的准备工作：

1. 自 ps 产生的输出如下：

```
    USER COMMAND
    root sched
    root /etc/init
    root /usr/lib/nfs/nfsd
...
   jones dtfile
   daemon /usr/lib/nfs/statd
...
```

2. sed 命令删除初始的标头行。
3. egrep 命令选定要被显示的 username。我们会清除 EGREP_OPTIONS 环境变量，以避免在不同的 GNU egrep 版本解译方式的冲突。
4. sort 先以 username，再以进程，进行排序数据。
5. uniq 命令附加前置重复行的计数，并删除重复部分。
6. 第二个 sort 步骤，再作一次数据排序。这次先以 username，再以由大而小的计数，最后才是进程名称作排序。
7. awk 命令将数据格式化为整齐的栏位，并删除重复的 username。

13.3 进程控制与删除

正常行为的进程，最终会如常地完成工作，再以 exit() 系统调用而终止。有时需要提早结束进程，可能是因为它一开始执行时便有误，也可能是需要你提供更多的资源才能正常执行，或是行为模式不对。

kill 命令的功能就在这，不过它的名字取得不好。kill 其实是传送信号（signal）给指定的执行中程序，不过有两个例外，这稍后会提到。进程接到信号，并处理之，有时可能直接选择忽略它们。只有进程的拥有者、或 root、内核、进程本身，可以传送信号给它。但接收信号的进程本身无法判断信号从何而来。

ISO Standard C 只定义六、七种信号类型，但 POSIX 增加了 20 多种，大部分系统还有更多，提供 30 至 50 种不同的信号。你可以这样列出它们，以下为 SGI IRIX 系统下的范例：

```
$ kill -l                          列出支持的信号名称（选项为小写的 L）
HUP INT QUIT ILL TRAP ABRT EMT FPE KILL BUS SEGV SYS PIPE ALRM TERM
USR1 USR2 CHLD PWR WINCH URG POLL STOP TSTP CONT TTIN TTOU VTALRM PROF
XCPU XFSZ UME RTMIN RTMIN+1 RTMIN+2 RTMIN+3 RTMAX-3 RTMAX-2 RTMAX-1
RTMAX
```

这些大部分都是专业用法，我们已在本书 Shell 脚本的 `trap` 命令中介绍过几个了。

每个处理信号的程序，都可自由决定解译这些信号的方式。信号名称反应的是惯用性（conventions），而不是必须性（requirement），所以对不同的程序而言，信号所表示的意义也会稍有不同。

无法抓取的信号通常会导致中断，不过 STOP 与 TSTP 通常只是暂停进程，直到 CONT 的信号出现，要求它再继续执行。你应该使用 STOP 与 CONT，以延迟合法进程的执行，直到系统不忙的时候。像这样：

```
$ top                                     显示 top 的资源消耗情况
...
  PID USERNAME  THR PRI NICE  SIZE   RES STATE    TIME   CPU COMMAND
17787 johnson     9  58    0  125M  118M cpu/3  109:49 93.67% cruncher
...

$ kill -STOP 17787                         终止进程

$ sleep 36000 && kill -CONT 17787 &        十小时后恢复
```

13.3.1 删除进程

以删除进程来说，必须要知道的有四个信号：ABRT（中断）、HUP（搁置）、KILL，与 TERM（终结）。

有些程序会在离开前做些清除工作：它们通常将TERM信号解译为“快速地清除并离开”的意思。如果你未指定信号，则 `kill` 会送出此信号。ABRT 有点类似 TERM，不过它会抑制清除的操作，并产生进程内存影像的副本，将其置于核心，即 *program*.core 或 core.*PID* 中。

HUP 信号有点类似要求中止，但是对于很多的 daemon 来说，它时常表示进程应先停止现在正在作的事，然后准备处理新的工作，好像它重新被启动一样。例如，在你改变设置文件后，HUP 信号可令 daemon 重读设置文件。

有两个信号是没有任何进程可捕捉或忽略的：KILL 与 STOP。这两个信号一定会立即被传送。然而对休眠进程（注 2）而言，这根据 Shell 实例与操作系统而定，大部分的其他信号都只在进程醒着（wake up）的时候才传送。因此，你应该预期在递送信号时，事实上是会有延迟的。

注 2： 进程等待某个事件，例如 I/O 完成或时间的过期处于搁置状态，称之为休眠（sleep），且进程调度器认为此时它并非可执行状态。当事件最终发生时，进程会再次进入可调度以执行的状态，这时被称为叫醒（awake）。

一次传送多个信号时，它们的传送顺序及是否传送相同信号超过一次以上，是不可预知的。有些系统所提供的只是保证：至少传送一个信号。信号的处理方式，在各种UNIX平台上有很大的差异，只有最简单的信号用法是具可移植性的。

我们已介绍过用以暂停进程的STOP信号，KILL信号则是让进程立即中止。以惯例而言，你应该先送出HUP信号给进程，让进程有机会优雅地中止。如果它没有马上离开，再试试TERM信号。如果这么做还是无法离开，再使用最后手段KILL信号。下面是它们的使用范例。假设你正面临停滞不前的响应：执行top命令，看看发生了什么事而得到像这样的结果：

```
$ top                                    显示 top 的资源消耗情况
...
  PID USERNAME THR PRI NICE SIZE  RES STATE   TIME    CPU COMMAND
25094 stevens    1  48    0 456M 414M cpu   243:58 99.64% netscape
...
```

网页浏览器通常只需要相当少的CPU时间，所以上述情况看起来此进程已经失控。我们传送HUP信号给该进程：

```
$ kill -HUP 25094                          传送 HUP 信号给进程 25094
```

再执行一次top，如果仍发现它还是没有马上消失在显示结果上，则使用：

```
$ kill -TERM 25094                       传送 TERM 信号给进程 25094
```

或最后的手段：

```
$ kill -KILL 25094                       传送 KILL 信号给进程 25094
```

大部分的top实例，会允许从top本身里，下达kill命令。

当然，只有你是stevens或root时才能作这样的事。否则，你只能要求系统管理员删除这个偏离的进程。

小心地使用kill命令。当程序不正常中止时，可能会在文件系统里留下残余数据。这些数据本应删除，除了浪费空间外，可能还会导致在下次执行程序时发生问题。例如：daemon、邮件客户端程序、文字编辑器，以及网页浏览器都会产生锁定（lock），其仅为一个小型文件，记录程序正在执行。如果程序的第二个实例（instance）被起动，而第一个实例仍在执行时，第二个实例会侦测到已存在的lock，回报该事实并立即中止。否则，两个实例写入同一个文件，将可能发生难以挽救的局面。糟糕的是，这些程序很少会告诉你lock文件的文件名，并很少将它写入文件里。如果该lock文件为长期执行进程的残余数据，你可能会发现程序无法执行，直到你找到lock并删除它为止。我们会在13.4节里告诉你怎么做。

有些系统（GNU/Linux、NetBSD，与 Sun Solaris）提供了 pgrep 与 pkill 命令，让你以名称追踪并删除进程。如未提供额外命令行选项，则 pkill 会传送一信号给“所有”指定名称的进程。以偏离的进程为例，我们会这么做：

```
$ pgrep netscape                        寻找netscape工作的进程编号
25094
```

接着：

```
$ pkill -HUP netscape                   传送HUP信号予netscape进程
$ pkill -TERM netscape                  传送TERM信号予netscape进程
$ pkill -KILL netscape                  传送KILL信号予netscape进程
```

不过，由于进程名称不是唯一的，因此以名称来删除它们会有风险：可能一次删掉太多，包括你不想删的。

13.3.2 捕捉进程信号

进程会向内核注册那些它们想要处理的信号。它们标明在signal()程序库调用的参数里，无论信号是否应该被抓取、忽略，或中止进程。为了令大部分程序无须烦恼这些信号的处理，内核本身即拥有一些信号默认值。例如，在Sun Solaris的系统上，我们发现：

```
$ man -a signal                          查看所有关于信号的手册页
...
     Name            Value    Default     Event
     SIGHUP          1        Exit        Hangup (see termio(7I))
     SIGINT          2        Exit        Interrupt (see termio(7I))
     SIGQUIT         3        Core        Quit (see termio(7I))
...
     SIGABRT         6        Core        Abort
...
     SIGFPE          8        Core        Arithmetic Exception
...
     SIGPIPE         13       Exit        Broken Pipe
...
     SIGUSR1         16       Exit        User Signal 1
     SIGUSR2         17       Exit        User Signal 2
     SIGCHLD         18       Ignore      Child Status Changed
...
```

trap 可引起 Shell 注册信号处理器（signal handler），抓取指定的信号。trap 取得一个字符串参数，其包含采取捕捉时要被执行的命令列表，紧接着一个要设置捕捉的信号列表。在旧式 Shell 脚本里，你会常看见以数字表示的信号。这不但无法让人了解它的用意，也不具可移植性，所以请使用信号名称。

例13-3展示的小型Shell脚本：looper，它的功能是使用trap命令，说明被抓取(caught)与未被抓取（uncaught）的信号。

例 13-3：休眠循环脚本：looper

```
#! /bin/sh -

trap 'echo Ignoring HUP ...' HUP
trap 'echo Terminating on USR1 ... ; exit 1' USR1

while true
do
    sleep 2
    date >/dev/null
done
```

looper里有两个trap命令，第一个只是回报HUP信号已被收到，而第二个则回报USR1信号并离开。之后，程序即进入休眠操作的无穷循环。我们将之执行于后台中，然后传送两个它要处理的信号：

```
$ ./looper &                                于后台执行 looper
[1]      24179                              进程 ID 为 24179

$ kill -HUP 24179                           传送 HUP 信号予 looper
Ignoring HUP ...

$ kill -USR1 24179                          传送 USR1 信号予 looper
Terminating on USR1 ...
[1] +  Done(1)                  ./looper &
```

现在来试试其他信号：

```
$ ./looper &                                再次于后台中执行 looper
[1]      24286

$ kill -CHLD 24286                          传送 CHLD 信号给 looper

$ jobs                                      looper 是否仍在执行中?
[1] +  Running                  ./looper &

$ kill -FPE 24286                           传送 FPE 信号给 looper
[1] + Arithmetic Exception(coredump)./looper &

$ ./looper &                                再次于后台执行 looper
[1]      24395

$ kill -PIPE 24395                          传送 PIPE 信号给 looper
[1] + Broken Pipe               ./looper &

$ ./looper &                                再次于后台执行 looper
[1]      24621

$ kill 24621                                传送默认信号 TERM 给 looper
[1] +  Done(208)                ./looper &
```

注意：CHLD 信号并未终止进程；它是内核里默认要被忽略的信号之一。相对地，符点

例外（floating-point exception，FPE）与停止管道（broken pipe，PIPE）信号则会引起进程中止。

再加入一个 trap 命令至 looper 作最后的试验：

```
trap 'echo Child terminated ...' CHLD
```

我们将修改过的脚本给予新的名称，再执行它：

```
$ ./looper-2 &                                于后台中执行 looper-2
[1]     24668
Child terminated ...
Child terminated ...
Child terminated ...
Child terminated ...

$ kill -ABRT 24668                            传送 ABRT 信号给 looper-2
[1] + Abort(coredump)          ./looper-2 &
```

每次循环主体的 sleep 与 date 进程中止时，都会取得 CHLD 捕捉，并大约每秒产生一次报告，直到我们传送 ABRT（中断）信号才终止循环进程。

除了先前 kill -l 所列的标准信号之外，Shell 另提供一个额外的信号供 trap 命令使用：EXIT。此信号数值恒被指定为零，所以 trap '...' 0 语句，在旧式的 Shell 脚本里等同于 trap '...' EXIT。

trap '...' EXIT 语句的本体，是在做 exit() 系统调用之前被引用，不是明确的 exit 命令，就是脚本的正常终止。如果为其他信号而设置捕捉，则这些捕捉会在 EXIT 的捕捉之前被处理。

执行 EXIT 捕捉时，离开状态 $? 的值会在捕捉完成时被保留下来，除非捕捉里的 exit 重设它的值。

bash、ksh，与 zsh 另提供两个给 trap 使用的信号：用以捕捉每个语句的 DEBUG，以及捕捉在语句之后回传的非零值离开码的 ERR。

DEBUG 捕捉就有点棘手了：在 ksh88 下，它是在语句之后捕捉，而后期的 Shell，则是在之前捕捉。公众软件的 Korn Shell 实例虽可在很多平台上使用，但完全不支持 DEBUG 捕捉。我们以下面的简短测试脚本说明它们的不同：

```
$ cat debug-trap                              显示测试脚本
trap 'echo This is an EXIT trap' EXIT
trap 'echo This is a DEBUG trap' DEBUG
pwd
pwd
```

在 Sun Solaris 系统下，我们使用几个不同的 Shell 测试此脚本：

```
$ /bin/sh debug-trap                        试试 Bourne Shell
test-debug-trap: trap: bad trap
/tmp
/tmp
This is an EXIT trap

$ /bin/ksh debug-trap                       试试 1988 (i) Korn Shell
/tmp
This is a DEBUG trap
/tmp
This is a DEBUG trap
This is an EXIT trap

$ /usr/xpg4/bin/sh debug-trap               试试 POSIX Shell (1988 (i) Korn Shell)
/tmp
This is a DEBUG trap
/tmp
This is a DEBUG trap
This is an EXIT trap

$ /usr/dt/bin/dtksh debug-trap              试试 1993 (d) Korn Shell
This is a DEBUG trap
/tmp
This is a DEBUG trap
/tmp
This is a DEBUG trap
This is an EXIT trap

$ /usr/local/bin/ksh93 debug-trap           试试 1993 (o+) Korn Shell
This is a DEBUG trap
/tmp
This is a DEBUG trap
/tmp
This is a DEBUG trap
This is an EXIT trap

$ /usr/local/bin/bash debug-trap            试试 GNU Bourne-Again Shell
This is a DEBUG trap
/tmp
This is a DEBUG trap
/tmp
This is a DEBUG trap
This is an EXIT trap

$ /usr/local/bin/pdksh debug-trap           试试公众软件的 Korn Shell
test-debug-trap[2]: trap: bad signal DEBUG

$ /usr/local/bin/zsh debug-trap             试试 Z-Shell
This is a DEBUG trap
/tmp
This is a DEBUG trap
/tmp
This is a DEBUG trap
```

```
This is an EXIT trap
This is a DEBUG trap
```

我们发现旧版bash与ksh的行为模式与这里测试出来的不太一样。简而言之，DEBUG捕捉产生的行为模式各有不同。这会是个问题，因此你不太可能在必须提供可移植性的Shell脚本里使用此捕捉。

ERR捕捉一样有出人意料的行为：命令替换失败时，则不捕捉。举例如下：

```
$ cat err-trap                                    显示测试程序
#! /bin/ksh -
trap 'echo This is an ERR trap.' ERR
echo Try command substitution: $(ls no-such-file)
echo Try a standalone command:
ls no-such-file

$ ./err-trap                                      执行测试程序
ls: no-such-file: No such file or directory
Try command substitution:
Try a standalone command:
ls: no-such-file: No such file or directory
This is an ERR trap.
```

两个ls命令都失败，但只有第二个会引发捕捉操作。

在Shell脚本里，最常使用的信号捕捉是脚本终结时的清理操作，像是删除暂时性文件。这类的trap命令引用，传统上会出现在Shell脚本的起始处附近：

```
trap ' 清理操作出现在此 ' EXIT
```

将捕捉设置在Shell的EXIT信号处通常就够用了，因为它会在所有其他的信号之后才被处理。在实际上，HUP、INT、QUIT与TERM信号也时常被捕捉。

如果要寻找更多在Shell脚本里使用捕捉的范例，你可以这么做：

```
grep '^trap' /usr/bin/*                寻找系统 Shell 脚本里的捕捉
```

我们发现许多脚本仍使用旧式的信号编号方式。signal()函数的使用手册通常会说明编号与名称的对应。

13.4 进程系统调用的追踪

很多系统都提供系统调用追踪器（system call tracers），它是在执行目标程序时，显示每个系统调用及目标程序执行时的参数。很可能你的系统里就有一个这样的程序，你可以试试以下列命令：ktrace、par、strace、trace或truss。虽然这些工具通常不会用在Shell脚本里，但它们可以帮助你找出进程正在做的事，还有为什么花了这么久的

时间。除此之外，它们无须访问源代码或改变你程序的使用方式，所以你可以将它们套用在任何属于你的进程上。此也有助于你对进程的了解，所以本节稍后会作些介绍。

如果你对UNIX系统调用的名称不熟悉，可以通过检查追踪日志迅速找到它们。它们的文件传统上是放在在线使用手册的Section 2；例如*open*(2)。例如，文件存在的测试通常会包含`access()`或`stat()`系统调用，而文件删除则需要`unlink()`系统调用。

大部分编译式程序语言都有除虫程序，允许使用单一步骤、设置中断点、变量检查等等。大部分系统上，Shell没有除虫程序，所以有时你得使用Shell的`-v`选项，显示Shell的输入行，或使用`-x`选项显示命令及其参数。系统调用追踪器对于输出结果提供了很有用的补充，因为它们可以让你更深入了解Shell引用的进程。

当你执行未知程序时，就表示你所做的这件事对系统可能造成危险。计算机病毒与蠕虫经常是以此方式散布。商用软件多半会随附安装程序，这是用户可以信任并执行的，有时甚至得要`root`权限才行。如果程序为Shell脚本，你便能进入一窥究竟。但如果它是像黑盒子一般的二进制影像文件，你就无从得知它的行为了。这类程序常会让用户觉得不安，我们多半不会以`root`的身份执行它。这时，如此一个安装的系统调用追踪日志就很有用了，它可以帮助你找出安装程序究竟做了些什么。就算你太晚知道而无法回复已删除的或已更改的文件，至少你可以知道哪些文件已受到影响，如果你的文件系统备份或快照（注3）还在，便能马上修复此灾难。

大部分长期执行的进程都会有许多系统调用，其追踪的输出可能会是很庞大的数量，因此，最好将之记录于文件。如果你只对几个系统调用有兴趣，你可以在命令行选项里指定它们。

我们现在来看看，GNU/Linux系统下建立的进程，追踪Bourne Shell的通信期。这可能会有点令人混淆，因为输出的来源有三：追踪（trace）、Shell以及我们所执行的命令。因此，我们设置提示号变量PS1，以兹区别原始与被追踪的Shell，这么一来，便能在每一行评注上它的来源了。`trace=process`参数会选定与进程相关的一群系统调用：

```
$ PS1='traced-sh$ ' strace -e trace=process /bin/sh     追踪与进程相关的系统调用
execve("/bin/sh", ["/bin/sh"], [/* 81 vars */]) = 0     追踪的输出
```

现在执行内置命令：

```
traced-sh$ pwd                                          执行Shell内置命令
/home/jones/book                                        这是命令输出
```

注3：　快照（snapshot）是近期某些高级文件系统的功能：它们可冻结文件系统的状态，通常只需数秒，保留当下目录树状结构的样式，用以在日后有所变动或发生问题时，根据此快照回复。

只有预期的输出会出现，因为没有新进程被建立。现在为该命令使用另一个程序：

```
traced-sh$ /bin/pwd                                                    执行外部命令
fork() = 32390                                                         这是追踪输出
wait4(-1,                                                              这是追踪输出
/home/jones/book                                                       这是命令输出
 [WIFEXITED(s) && WEXITSTATUS(s) =  = 0], WUNTRACED, NULL) = 32390 这是追踪输出
--- SIGCHLD (Child exited) ---                                         这是追踪输出
```

最后，离开 Shell，追踪即为：

```
traced-sh$ exit                                                        自 Shell 离开
exit                                                                   这是追踪输出
_exit(0) = ?                                                           这是追踪输出
```

现在回到原始的 Shell 通信期：

```
$ pwd                                      回到原始的 Shell，确认我们所在位置
/home/jones/book                           工作中目录未变更
```

Shell 发出了 fork() 系统调用，以启动 /bin/pwd 进程，其输出与下一个 wait4() 系统调用的追踪报告混合在一起。命令如常终止，且 Shell 收到 CHLD 信号，指出子进程完成。

下列为 Sun Solaris 上探测系统调用的范例。-c 选项要求在命令完成后显示摘要报告，抑制追踪报告的一般输出：

```
$ truss -c /usr/local/bin/pathfind -a PATH truss       追踪 pathfind 命令
/usr/bin/truss                                         这是 pathfind 产生的输出
/bin/truss
/usr/5bin/truss
syscall               seconds   calls  errors           truss 的报告由此开始
_exit                     .00       1
fork                      .00       2
read                      .00      26
write                     .00       3
open                      .00       5       1
close                     .00      10       1
brk                       .00      42
stat                      .01      19      15
...
stat64                    .03      33      28
open64                    .00       1
                      -------  ------    ----
sys totals:               .04     242      50
usr time:                 .01
elapsed:                  .19
```

当程序执行时间超出你所预期时，类似上述的输出就能帮助你，可以通过系统调用找出执行效能上的瓶颈。time 命令可以为系统调用探测识别出候选人：它会报告用户时间、系统调用时间及墙上时钟时间。

注意： 监控文件访问最常见的系统调用追踪的应用是：可以在追踪日志中寻找access()、open()、stat()与unlink()的调用报告。在GNU/Linux上，使用strace -e trace=file可减少日志量。当全新安装的软件，抱怨找不到所需的组态文件，又无法告诉你文件名称时，文件访问追踪就派得上用场了。

系统调用追踪器对于寻找我们先前所提及的残留锁定文件也很有用。下面是在Sun Solaris系统上，如何找出由特定网页浏览器所产生的锁定文件：

```
$ truss -f -o foo.log mozilla                 追踪浏览器执行
$ grep -i lock foo.log                        查找追踪文件里的单词“lock”
...
29028:  symlink("192.168.253.187:29028",
                "/home/jones/.mozilla/jones/c7rboyyz.slt/lock") = 0
...
29028:  unlink("/home/jones/.mozilla/jones/c7rboyyz.slt/lock") = 0
```

此浏览器产生的锁定文件，指向一个不存在文件名的符号性连接，此文件名包含本地端机器的数值型Internet主机位置及进程编号。当浏览器进程提早死亡时，删除锁定文件的unlink()系统调用便不会被执行。锁定文件名不见得总是有lock这个字在其中，所以有时你可能得更仔细审视追踪日志，找出你要的锁定文件。

此处为SGI IRIX系统上缩简的追踪情况，我们要测试的是/bin/sh是否可执行：

```
$ /usr/sbin/par /bin/test -x /bin/sh          追踪 test 命令
...
    0mS[  0] : execve("/bin/test", 0x7ffb7e88, 0x7ffb7e98)
...
    6mS[  0] : access("/bin/sh", X_OK) OK
    6mS[  0] : stat("/bin/sh", 0x7ffb7cd0) OK
...
    6mS[  0] : prctl(PR_LASTSHEXIT) = 1
    6mS[  0] : exit(0)
System call summary :
                          Average     Total
Name             #Calls  Time(ms)   Time(ms)
-----------------------------------------
execve                1      3.91       3.91
open                  2      0.11       0.21
access                1      0.17       0.17
stat                  1      0.12       0.12
...
prctl                 1      0.01       0.01
exit                  1      0.00       0.00
```

当你找到你要的系统调用就可以限制追踪输出，只显示特定的调用，让画面不致太过混乱：

```
$ /usr/sbin/par -n stat /bin/test -x /bin/sh          仅追踪 stat 系统调用
    0mS[  0] (5399999) : was sent signal SIGUSR1
    0mS[  3] : received signal SIGUSR1 (handler 0x100029d8)
    6mS[  3] : stat("/bin/sh", 0x7ffb7cd0) OK

System call summary :
...
```

BSD与Mac OS X的ktrace命令的运行有点不太一样，它们是将追踪结果写成二进制文件：ktrace.out。之后，再执行kdump将其转换为文字形式。下面是来自NetBSD系统的追踪，测试/bin/sh的执行权限：

```
$ ktrace test -x /bin/sh                    追踪 test 命令

$ ls -l ktrace.out                          列出追踪日志
-rw-rw-r--    1 jones    devel    8698 Jul 27 09:44 ktrace.out

$ kdump                                     追踪日志的后续处理
...
 19798 ktrace   EMUL   "netbsd"
 19798 ktrace   CALL   execve(0xbfbfc650,0xbfbfcb24,0xbfbfcb34)
 19798 ktrace   NAMI   "/usr/local/bin/test"
...
 19798 test     CALL   access(0xbfbfcc80,0x1)
 19798 test     NAMI   "/bin/sh"
 19798 test     RET    access 0
 19798 test     CALL   exit(0)
```

追踪日志还得作后续处理实在不是很理想，因为这会妨碍我们得知进程所作的系统调用之动态情况。而且大型的系统调用可能更难以识别。

所有的系统调用追踪器都能以进程ID作为参数，而非命令名称，这使得它们可以追踪已经在执行的进程。但只有进程拥有者与root可以作这件事。

我们这里说明的只是一小部分，其实能用的系统调用追踪器比我们这里提到的多很多。请参考你本地端机器的使用手册，了解更多细节。

13.5 进程账

UNIX系统支持进程账（process accounting）功能，不过为减少管理性日志文件的负荷，此功能通常停用。该功能启用时，每当一个进程完成，内核便会写入一个简洁的二进制记录到系统相依的账目文件里，例如/var/adm/pacct或/var/account/pacct。账目文件在转为文字数据流前，必须先作处理。例如，在Sun Solaris上，root可能得这么做才能产生我们看得懂的列表：

```
# acctcom -a                                列出账目记录
...
```

```
COMMAND                      START    END        REAL    CPU     MEAN
NAME        USER TTYNAME TIME     TIME       (SECS)  (SECS)  SIZE(K)
...
cat         jones       ?  21:33:38 21:33:38   0.07    0.04 1046.00
echo        jones       ?  21:33:38 21:33:38   0.13    0.04  884.00
make        jones       ?  21:33:38 21:33:38   0.53    0.05 1048.00
grep        jones       ?  21:33:38 21:33:38   0.14    0.03  840.00
bash        jones       ?  21:33:38 21:33:38   0.55    0.02 1592.00
....
```

由于各UNIX之间的输出格式与账目工具之实例都有所不同，因此我们无法为这种摘要性账目数据提供可移植性的脚本。不过，样本输出显示文字格式相对简单许多。例如，我们可以轻松地产生前十名的命令列表及使用量计数：

```
# acctcom -a | cut -d ' ' -f 1 | sort | uniq -c | sort -k1nr -k2 | head -n 10
21129 bash
5538 cat
4669 rm
3538 sed
1713 acomp
1378 cc
1252 cg
1252 iropt
1172 uname
 808 gawk
```

这里，我们使用cut摘取出第一栏，再以sort排序此列表，以uniq减少重复的计数，再重新由大至小排序计数，最后使用head在列表里显示前十笔数据。

使用apropos accounting命令，可找出你系统里的账目命令。常见的有acctcom、lastcomm与sa：它们都有很多选项可用以简化大量日志数据，使其成为易于管理的报告。

13.6 延迟的进程调度

大部分时候，用户都是希望进程马上起始，快点结束。而Shell的执行，也是在前一个命令后，马上接着执行下一个命令。命令完成的速度是与资源的限制有关，且不在Shell的权限下。

在交谈模式使用下，有时不必等到命令完成才能执行另一个。这是Shell提供的一个简单方式：所有的命令只要在最后加上&字符，都可起始于后台执行，无须等待。只有在少数情况下，必须等待后台进程完成，见13.2节里所提及的wait命令。

至少有4种情况需要延迟进程起始，直到未来的某个时间点才执行，我们将于以下子节介绍。

13.6.1 sleep：延迟片刻

当进程应于某个特定时间过后才能启动时，可使用sleep命令暂停执行一段指定的秒数之后，再下达被延迟的命令。sleep使用的资源很少，且可在不会对活动中的进程有任何干扰下被使用。事实上，调度器只是忽略休眠中的进程，直到其计时器届满才叫醒它们。

在例13-1与例13-3我们都使用短暂的Sleep建立一个无穷循环的程序，但这么做并不会耗掉系统的太多资源。在9.10节中的短暂Sleep用以确保循环中为各个进程选择一个新虚拟随机产生器种子。在13.3节里的长时间Sleep，等待一段时间，直到系统有更多时间处理时，才重启消耗大量资源的工作。

大部分deamon在执行其工作时，都会短暂休眠之后再醒来检查更多的工作。在这种方式下，它们只会耗用少数资源，执行时对其他进程的影响很小。它们通常是引用sleep()或usleep()函数（注4），反而不会直接使用Sleep命令，除非它们本身就是Shell脚本。

13.6.2 at：延迟至特定时间

at命令可以令程序在特定时间执行。该命令语法在系统间各异，不过下面的例子为普遍形式：

```
at 21:00                < command-file  在下午 9 点执行
at now                  < command-file  马上执行
at now + 10 minutes     < command-file  10 分钟后执行
at now + 8 hours        < command-file  8 小时后执行
at 0400 tomorrow        < command-file  明天早上 4 点执行
at 14 July              < command-file  在下一个国庆日 (Bastille Day) 执行
at noon + 15 minutes    < command-file  在今天的 12:15 执行
at teatime              < command-file  在今天下午执行
```

上述每个例子，要执行的工作都定义在*command-file*里的命令。at指定时间的方式有点哲学，像最后一个例子指的就是16:00。

atq列出at队列里的所有工作，而atrm则是删除它们。欲了解更进一步的细节，请参考你系统里的at使用手册。

注意： 部分系统上，用以执行at的Shell为Bourne Shell（/bin/sh），而你登录的Shell可能又是在其他系统上。你可以避开这些不稳定的情况，在at单行命令输入时，指定一个你觉得好用的语言所写的可执行脚本，首行设置如下：

```
#! /path/to/script/interpreter
```

注4： 不同的系统会有所不同，有些是系统调用，有些是程序库函数。

at命令系列能否可用根据管理政策而定。at.allow与at.deny两个文件即用以控制其访问：它们可能是储存在/etc、/usr/lib/cron/at、/var/adm/cron或/var/at中，根据UNIX版本而定。假如找不到这两个文件，那就只有root可以使用at了。如果你的系统不允许你使用at命令，你可以向你的系统管理员反映，大部分站台，没有理由禁止这个功能。

13.6.3 batch: 为资源控制而延迟

在计算机提供互动模式给人们访问之前，操作系统执行所有的进程都是采取批处理模式(batch mode)。工作数据流的执行是累积式的，而它们的执行顺序可能视工作在队列里的位置、你的身份、你的重要性、你所需要的资源与你能拥有的资源、你要准备等待多久，以及你愿意付出多少而定。许多大型主机以及大型计算服务器，仍以此方式来耗用其CPU周期。

现行所有UNIX系统都支持batch命令，让你将进程加入至某个批处理队列中。batch的语法在系统间各异，不过它们都支持读取来自标准输入的命令：

```
batch < command-file                    批处理执行命令
```

某些系统下，它等同于：

```
at -q b -m now < command-file           立即执行在批处理队列下的命令
```

其中，-q b为指定批处理队列，-m则是要求在工作完成时寄送邮件予用户，而now意即已准备好立即执行。

batch的问题便是它太简化了：对批处理处理的顺序提供的控制很少，也没有一套批处理政策。这个命令很少用在小型系统上；而在大型系统上，特别是那些分散式系统，batch也已被许多更复杂的实例所取代，像我们在表13-2所列的那些。这些包都提供整套完整的命令，用来委任与管理批处理工作。

表13-2：进阶的批处理队列与调度系统

名称	网站
Generic Network Queueing System	*http://www.gnqs.org/*
IBM LoadLeveler	*http://www.ibm.com/servers/eserver/pseries/library/sp_books/loadleveler.html*
Maui Cluster Scheduler	*http://supercluster.org/maui/*
Platform LSF system	*http://www.platform.com/products/LSFfamily/*
Portable Batch System	*http://www.openpbs.org/*

表 13-2：进阶的批处理队列与调度系统（续）

名称	网站
Silver Grid Scheduler	*http://supercluster.org/silver/*
Sun GridEngine	*http://gridengine.sunsource.net/*

13.6.4 crontab：在指定时间再执行

大部分计算机有许多管理工作需要重复执行，像是每晚的文件系统备份、每周的日志文件与暂时性目录的清空、每月的账目报表等等。一般用户也会需要用到这样的功能，例如将办公室计算机里的文件与家里计算机中的文件进行同步。

这个工具程序可在指定的时间执行工作，其包括了在系统启动时起始的 cron daemon，与 crontab 命令，此命令用来管理/记录工作应何时执行的简单文字文件：见 *cron*(8)与 *crontab*(1)的使用手册。你可以通过 crontab -l（小写字母 L）列出你的目前工作调度，以 crontab -e 启动编辑器更新调度。编辑器的选择根据 EDITOR 环境变量而定，有些计算机会因为未设置此参数而拒绝执行 crontab，但有些则会直接启动 ed。

crontab 文件（见 *crontab*(5)使用手册）支持 Shell 式注释，所以在它的起始处具有说明是很有用的，可提醒我们其语法：

```
$ crontab -l                                    列出目前的 crontab 调度
#        mm     hh     dd     mon     weekday       command
#        00-59  00-23  01-31  01-12   0-6(0=Sunday)
...
```

前 5 个栏位，除了使用单一数字外，还可以搭配连字符分隔，指出一段（含）区间（例如，第二个栏位里的 8-17，指的是 08:00 至 17:00 间每小时执行一次），或者使用逗点分隔数字列表或区间（例如：第一个字段的 0,20,40 指的便是每 20 分钟执行一次）。你还可以使用星号，指该字段所有可能的数字，见下面范例：

```
15  *    *  *   *    command         每个小时的第 15 分钟执行
 0  2    1  *   *    command         每个月一开始的 02:00 执行
 0  8    1 1,7  *    command         一月一日与七月一日的 08:00 执行
 0  6    *  *   1    command         每周一的 06:00 执行
 0 8-17  *  *  0,6   command         每周末的 08:00 至 17:00 间，一小时执行一次
```

警告： 虽然 POSIX 宣称空白行会忽略不处理，但有些商用的 crontab 版本却无法容许这样的情况，还会确实地删除含有空白行的 crontab 文件！所以我们建议你，不要在你的 crontab 文件里置入空白行。

crontab文件里的命令执行于一些已经设置的环境变量下：SHELL为/bin/sh，但HOME、LOGNAME，有时还包括USER，则是根据passwed文件或数据库里的记录值而被设置。

PATH的设置极为严格，通常只设置/usr/bin。如果你习惯更自由的设置，则你要不就得在crontab文件中指定命令的完整路径名称，要不就是明白地设置PATH：

```
0 4 * * * /usr/local/bin/updatedb                     每夜更新GNU的快速查找数据库
0 4 * * * PATH=/usr/local/bin:$PATH updatedb          类似上述操作，但将PATH传递予
                                                      updatedb的子进程
```

任何出现在标准错误输出或标准输出上的数据都会寄给你，或是在其他实例中，会寄到MAILTO变量的值所指定的用户。实务上，你通常会比较倾向于将输出重导至一个日志文件，并累积连续执行的记录。crontab实例记录会有点类似这样：

```
55 23 * * *    $HOME/bin/daily >> $HOME/logs/daily.log 2>&1
```

此类日志文件会持续成长，所以你应该适时地清除它，你可以使用编辑器删除此日志文件的前半段，也可以使用tail -n *n*取出最后的*n*行：

```
cd $HOME/logs                                 切换至日志文件所在目录
mv daily.log daily.tmp                        重新命名日志文件
tail -n 500 daily.tmp > daily.log             回复最后的 500 行
rm daily.tmp                                  舍弃旧的日志文件
```

在做这个操作时，只要确认日志文件当时未正在进行更新操作即可。很明显地，对于这个必须重复不断执行的进程，我们可以，也应该这么做，将它委托给另一个crontab实例记录。

累积的日志文件另一个好用的替代方案就是加上时间戳记的文件，让每个cron工作日志都拥有一个文件。以每日的日志来说，我们可以使用crontab项目如下：

```
55 23 * * *  $HOME/bin/daily > $HOME/logs/daily.`date +\%Y.\%m.\%d`.log 2>&1
```

cron通常会将命令行上的百分比字符变更为换行符号，不过加上反斜线即可避免这样的不寻常行为模式。

你也可以很轻松地压缩或删除旧的文件，只要通过find命令即可：

```
find $HOME/logs/*.log -ctime +31 | xargs bzip2 -9     压缩一个月前的日志文件

find $HOME/logs/*.log -ctime +31 | xargs rm           删除一个月前的日志文件
```

注意： 为保持crontab文件的简明，将它的每个命令以设计好的文件名，分别存放在个别的Shell脚本中。可供你稍后修订这些脚本，而无须动到你的crontab文件。

如果执行cron工作的第二个实例可能会造成伤害的话（例如文件系统备份或日志文件更新），你就必须避免这种情况，要不就是得使用适当的锁定文件，要不就是于工作本身结束之前把cron换成at。当然，你之后必须监控它每次执行后的情况。这么一来，当发生错误事件时，如果你使用的是锁定文件，你必须确定已删除它们；如果你使用的是at，则得重新调度工作。

你可以使用crontab -r将crontab文件整个删除。它就像rm一样无法撤回，也无法复原。我们建议你保留备份，像这样：

```
crontab -l > $HOME/.crontab.`hostname`     储存现行crontab
crontab -r                                 删除crontab
```

这么一来，之后你可以如此这般的回复：

```
crontab $HOME/.crontab.`hostname`          回复储存的crontab
```

由于这里假设一个主机里只有一个crontab文件，所以我们将主机名称包括在被储存的文件名称里，这么一来便能马上识别出它是属于哪台主机的文件。

crontab会以命令行上给定的文件置换任何已存在的调度，提供的语法必须正确无误，否则将保留旧的调度。

就像at命令那样，系统目录里也有cron.allow与cron.deny文件，用以控制是否允许cron工作，以及谁可以执行它们。如果你发现你不能使用这个好用的工具，请向系统管理员反映。

13.7 /proc文件系统

有些UNIX版本，借用了贝尔实验室所开发的想法：/proc文件系统。与其通过无数的系统调用不断地更新，以提供内核数据的访问，不如通过一个特殊文件设备，访问内核里的数据，也就是在/proc目录内实例一个标准的文件系统界面。每个执行中的进程都会在那里拥有一个子目录，以进程编号命名，且在每个子目录里面是各式各样小文件的内核数据。这个文件系统的内容，可参考*proc*(4)（大部分系统）或*proc*(5)（GNU/Linux）的使用手册。

GNU/Linux开发了比UNIX各类版本还多的此想法，且它的ps命令更能取得所有需要的进程数据，只要读取/proc下的文件即可。你可通过系统调用追踪strace -e trace=file ps aux的执行，来验证此说法。

下面的范例是一个文字编辑器通信期的进程文件：

```
$ ls /proc/16521                    为进程 16521 列出 proc 文件
cmdline  environ  fd    mem     root  statm
cwd      exe      maps  mounts  stat  status

$ ls -l /proc/16521                 以冗长式列表，再列出一次
total 0
-r--r--r--    1 jones    devel   0 Oct 28 11:38 cmdline
lrwxrwxrwx    1 jones    devel   0 Oct 28 11:38 cwd -> /home/jones
-r--------    1 jones    devel   0 Oct 28 11:38 environ
lrwxrwxrwx    1 jones    devel   0 Oct 28 11:38 exe -> /usr/bin/vi
dr-x------    2 jones    devel   0 Oct 28 11:38 fd
-r--r--r--    1 jones    devel   0 Oct 28 11:38 maps
-rw-------    1 jones    devel   0 Oct 28 11:38 mem
-r--r--r--    1 jones    devel   0 Oct 28 11:38 mounts
lrwxrwxrwx    1 jones    devel   0 Oct 28 11:38 root -> /
-r--r--r--    1 jones    devel   0 Oct 28 11:38 stat
-r--r--r--    1 jones    devel   0 Oct 28 11:38 statm
-r--r--r--    1 jones    devel   0 Oct 28 11:38 status
```

请注意，这里所有的文件似乎都是空的。但事实上，它们都包含了设备驱动程序在被读取时所提供的数据：它们从未真的存在于储存设备里。它们的时戳也是有疑问的：在GNU/Linux 与 OSF/1 系统下，它们反映的是目前时间，但在 IRIX 与 Solaris 里，它们显示的却是每个进程起始的时间。

/proc 文件的大小为零，造成部分工具程序的混淆，像是 scp 与 tar。你可能得先试着使用 cp，将它们复制到别处的一般文件里。

现在我们来看看其中一个文件：

```
$ cat -v /proc/16521/cmdline              显示进程命令行
vi^@+273^@ch13.xml^@
```

-v 选项将无法打印的字符以脱字符号注释来显示，^@ 所表示的就是 NUL 字符。显然地，此文件包含一连串以 NUL 终结的字符串，还有命令行里的参数。

除了特定进程数据之外，/proc 还包括其他有用的文件：

```
$ ls /proc | egrep -v '^[0-9]+$' | fmt    列出所有除了进程以外的目录
apm bus cmdline cpuinfo devices dma driver execdomains fb
filesystems fs ide interrupts iomem ioports irq isapnp kcore kmsg
ksyms loadavg locks mdstat meminfo misc modules mounts mtrr net
partitions pci scsi self slabinfo speakup stat swaps sys sysvipc
tty uptime version
```

这是其中一个的起始：

```
$ head -n 5 /proc/meminfo                 显示内存信息的前 5 行
```

```
         total:    used:    free:  shared: buffers:  cached:
Mem:  129228800 116523008 12705792        0  2084864 59027456
Swap: 2146787328 28037120 2118750208
MemTotal:        126200 kB
MemFree:          12408 kB
```

通过文件方式取得进程数据是很方便的，这也使得就算缺乏系统调用界面，数据也很容易通过程序语言所撰写的程序来取得。例如，Shell脚本可以从 `/proc/*info`文件收集CPU、内存，与储存设备在硬体方面的细节，前提当然是你的系统环境下有提供这样的文件，然后再产生类似 sysinfo（注5）命令所作的华丽报告。然而，这些文件缺乏标准化的内容，使得产生统一的报告更为困难。

13.8 小结

在本章，我们呈现了：如何建立、列出、控制、调度与删除进程，还有如何将信号传送给它，以及如何追踪它们的系统调用。由于进程执行于私有地址空间中，因此它们不会彼此干扰，也不需要特别花力气写程序让它们在同一时间执行。

进程都可捕捉所有的信号（只有两个例外），它们要不就是忽略它，要不就是响应期待的操作。无法捕捉的两个信号是KILL与STOP，都是为了确保如果有行为不当的进程，都可马上杀除或暂停之。需要执行清理操作的程序，像是储存活动中的文件、重设终端机模式，或是删除锁定，通常都会要捕捉一般信号；否则，绝大多数无法捕捉的信号，都会导致进程中止。有了trap命令，将简单的信号处理加入Shell脚本里就更容易了。

最后，我们检查各种不同的延迟与控制进程执行的机制。在这里面，sleep为撰写Shell脚本时最好用的一个，不过其他命令还是各有其不可或缺的用途。

注5： 可在 *http://www.magnicomp.com/sysinfo/* 取得。

第 14 章

Shell 可移植性议题与扩展

在 POSIX 下定义的 Shell 语言，比原始的 V7 Bourne Shell 规模大很多，但又比 ksh93 与 bash 所实例的语言小，这两种语言是 Bourne Shell 扩展版本里，最广泛使用的。

要编写一个耐用的脚本使用到 Shell 语言的所有扩展的优势，可能就会用到这两个 Shell 的其中一个，甚至是两个并用。因此，了解它们的共通功能及差异，绝对是很重要的。

长久下来，bash 取用许多 ksh93 里的扩展，但并非全部。所以它们有相当多功能是重叠的，却也有很多的地方不同。本章将大致描述 bash 与 ksh93 的差异以及它们共通的扩展，与 POSIX Shell 所提供的功能。

注意： 很多在这里提供的功能，都只有 ksh93 近期版本才能使用。一些商用 UNIX 系统仍使用旧版本的 ksh93，特别是 dtksh（desktop Korn Shell，/usr/dt/bin/dtksh）这个程序，就没有更新的功能可用。最好的方式应该是下载当前 ksh93 的原始码并从头开始建置，详见 14.4 节。

14.1 迷思

下面是须特别留意的几个项目：

存储 Shell 状态

例 14-1 告诉你如何存储 Shell 状态至文件中。但 POSIX 标准里有个失察之处：未定义存储函数定义的方式供日后回复之用！下面就是要告诉你，如何使用 bash 与 ksh93 完成此任务。

例 14-1：存储 bash 与 ksh93 的 Shell 状态，包括函数

```
{
  set +o                        选项设置
  (shopt -p) 2>/dev/null        bash 特定的选项，subShell 会使 ksh 沉默
  set                           变量与值
  export -p                     被导出的变量
  readonly -p                   只读变量
  trap                          捕捉设置

  typeset -f                    函数定义 (非 POSIX)
} > /tmp/Shell.state
```

要注意的一点是：bash与ksh93定义函数时使用的语法不同，所以如果你想在其中一个 Shell 输出状态，然后在另一个 Shell 里回复时，必须小心处理。

echo 不具可移植性

在 2.5.3 节里，曾说过 echo 命令只有最简单的用法可使用在必须具可移植性的脚本上，且其各种选项与转义符有些具可移植性有些又没有（尽管它的确在 POSIX 标准下）。

在 ksh93 下，内置的 echo 版本会试图模仿 $PATH 下所找到的外部版本 echo 之行为模式。此操作背后的理由就是为了兼容性：在所有UNIX系统上，当Korn Shell 执行 Bourne Shell 脚本时，它应当遵循与原 Bourne Shell 相同的行为模式。

另一方面，bash 下内置版本的行为模式在各 UNIX 系统间都是相同的。原理根据是一致性：bash脚本应有相同的行为模式，不应该因为执行于不同的 UNIX 系统而有所差异。因此为了完整的可移植性，echo应避免使用，printf仍是最好的选择。

OPTIND 可为本地端变量

在 6.4.4 节中我们提到过 getopts 命令、OPTIND 与 OPTARGS 变量，ksh93 提供了定义函数的本地端版本的OPTIND函数关键字。它的想法是：函数可以像个别的脚本一样使用getopts，以等同于脚本的方式处理它们的参数，而不影响其父选项的处理。

${*var*:?*message*}可能不存在

${*variable*:?*message*}变量展开会检查是否*variable*已设置。如果否，则Shell 显示*message*信息并离开。但如果Shell是在交互模式下，行为模式就不同了。盲目地离开对于交互模式Shell而言并非总是正确的，因为可能会让用户注销。下面的脚本名为 x.sh：

```
echo ${somevar: ?somevar is not set}
echo still running
```

针对上述脚本，bash 与 ksh93 的行为模式如表 14-1 所示。

表 14-1：${var:?message}在 bash 与 ksh93 下的互动

命令	显示信息	接下来命令的执行
$ **bash x.sh**	是	否
$ **ksh93 x.sh**	是	否
bash$. **x.sh**	是	是
ksh93$. **x.sh**	是	否

这里指出，如果你确定脚本可以用点号命令执行，则你应该确保它在使用 ${*variable*:?*message*} 架构之后便离开。

在 for 循环中，漏失循环项目

这部分很难解释，先看下面这个循环：

```
for i in $a $b $c
do
    执行一些操作
done
```

如果三个变量都为空，则没有值给循环处理，Shell 就静默地不作任何操作。就像这么写循环一样：

```
for i in          # 什么事也没做!
do
    执行一些操作
done
```

然而，对大部分的 Bourne Shell 版本来说，真的这么编写循环会产生语法错误。不过 2001 POSIX 标准中，已将直接进入一个空循环的作法制定为合法。

当前版本的 ksh93 与 bash 都接受空的 for 循环，只是静默地不做任何事。不过这是近期出现的功能，这两套 Shell 的旧版本以及原始的 Bourne Shell，可能还是会产生语法错误的信息。

DEBUG 捕捉，行为模式各异

ksh88 与 ksh93 都提供特殊的 DEBUG 捕捉，以利 Shell 除虫与追踪。在 ksh88 下，DEBUG 的捕捉是在每个命令执行之后发生，但在 ksh93 下，却是在命令执行之前发生。到当前为止还好。较让人觉得混淆的是：bash 早期版本遵循 ksh88 的行为模式，但现行版本却是遵循 ksh93。这点在 13.3.2 节里即已说明。

set 的长与短选项

两个 Shell 下的 set 命令都接受额外的短选项与长选项，它们完整的 set 选项列于表 14-2。标注 POSIX 的项目在 bash 与 Korn Shell 中都可使用。

表 14-2：set 的 Shell 选项

短选项	-o 形式	可用性	叙述
-a	allexport	POSIX	导出所有接续的已定义变量。
-A		ksh88，ksh93	数组指定。set +A 不会清除数组。详见 14.3.6 节。
-b	notify	POSIX	马上显示工作完成的信息，而不是等待下一个提示。为交互式使用所设计。
-B	braceexpand	bash	启用括弧展开。默认值是启用的。详见 14.3.4。
-C	noclobber	POSIX	不允许 > 重导至已存在的文件，但 >| 运算符会使此选项的设置无效。为交互式使用所设计。
-e	errexit	POSIX	当命令以非零状态离开而结束 Shell。
-f	noglob	POSIX	停用通配字符展开。
-h	hashall (bash) trackall (ksh)	POSIX	在函数被定义时，找出及记得自函数主体中调用命令的位置，而不是在函数执行时做这件事 (XSI)。
-H	histexpand	bash	启用 ! 风格的历史展开。默认值是启用的[注]。
-k	keyword	bash，ksh88，ksh93	将所有变量指定放进环境里，即便它们位居命令的中间。这是已过时的功能且不应该再使用。
-m	monitor	POSIX	启用工作控制（默认值是启用的）。为交互式使用所设计。
-n	noexec	POSIX	读取命令并检查是否有语法错误，但不要执行它们。交互模式下的 Shell 允许忽略此选项。
-p	privileged	bash，ksh88，ksh93	尝试在更安全的模式运作。细部的处理方式在各 Shell 间都有所不同，请参考你 Shell 的文件。
-P	physical	bash	使用实体的目录结构，执行改变目录的命令。
-s		ksh88，ksh93	排序位置型参数。
-t		bash，ksh88，ksh93	读取并执行命令然后离开。这是过时的用法，它是为了与 Bourne Shell 的兼容性且不应该再使用。
-u	nounset	POSIX	将未定义的变量视为错误，而非 null。
-v	verbose	POSIX	在命令执行之前，（逐字地）显示它们。
-x	xtrace	POSIX	在命令执行之前，显示它们（于展开之后）。
	bgnice	ksh88，ksh93	自动地降低所有后台执行（带有 &）之命令优先权。
	emacs	bash，ksh88，ksh93	使用 emacs 风格的命令列编辑。为交互式使用所设计。

表 14-2：set 的 Shell 选项（续）

短选项	-o 形式	可用性	叙述
	gmacs	ksh88，ksh93	使用GNU emacs风格的命令列编辑。为交互式使用所设计。
	history	bash	启用命令历史功能，默认值为启用的。
	ignoreeof	POSIX	停用 Ctrl-D 离开 Shell。
	markdirs	ksh88，ksh93	执行通配字符展开时，将 / 附加到目录。
	nolog	POSIX	停用对函数定义所执行的命令历史功能。
	pipefail	ksh93	令管道离开状态是上一个失败命令的状态，或是如运作无误，则为零。仅 ksh93n 或更新版适用。
	posix	bash	启用全 POSIX 兼容功能。
	vi	POSIX	使用vi式的命令列编辑。为交互式使用所设计。
	viraw	ksh88，ksh93	使用vi式的命令列编辑。为交互模式使用所设计。此模式较 set -o vi 更消耗使用 CPU。

注：若你使用 bash，我们建议关闭此功能。

14.2 bash 的 shopt 命令

bash Shell 除了可使用 set 命令的长选项与短选项之外，另还有 shopt 命令，可停用与启用选项。

bash 3.0 版的选项列表如下。我们会针对每个选项，叙述当它们设置时（启用时）的行为模式：

cdable_vars

当 cd 的参数不是目录时，bash 会将它视为变量名称，其值为目标目录。

cdspell

如果 cd 至目录的操作失败，bash 会试图进行多次微幅的拼字修正，看看是否能找到真正的目录。如果找到正确的，它会显示名称，并切换至经运算过的目录。此选项仅于交互式 Shell 下运作。

checkhash

当 bash 在路径查找之后找到命令，它会将路径查找的结果存储在杂凑表格中，此举可加速下一次相同命令的执行。当命令第二次被执行时，bash 便会执行存储于杂凑表格里的命令。有了这个选项，bash 会在执行它之前，先验证存储在杂凑表格中的文件名真的存在。如果找不到，bash 便会按照正规方式，进行路径查找。

shopt (bash)

语法

`shopt [ -pqsu ] [ -o ] [ option-name ... ]`

用途

当 Shell 选项加入至 bash 时，进行集中控制，而非增生的 set 选项或 Shell 变量。

主要选项

-o

限制选项为那些可以用 set -o 设置的。

-p

以适于重读的形式打印输出。

-q

静默模式。其离开状态可指出选项是否被设置。以多选项而言，如果它们都是启用的，则状态为零，否则即为非零。

-s

设置（启用）给定的选项。

-u

解除设置（停用）给定的选项。

如果 -s 与 -u 不带指名的选项，则分别显示设置或解除设置的选项列表。

行为模式

控制各种内部 Shell 选项的设置。未带选项或使用 -p 时，则显示设置。使用 -p，会以可供稍后重读的形式显示设置。

警告

仅 bash 适用，ksh 则否。

checkwinsize

在每个命令之后，bash 都会检查窗口大小，并于窗口大小变更时，更新 LINES 与 COLUMNS 变量。

cmdhist

bash将多行命令的所有行都存储于历史文件里。这使我们得以重新编辑多行命令。

dotglob

bash 在文件名展开的结果里包含名称开头为 .（点号）的文件。

execfail

如果 bash 无法执行 exec 内置命令所给定的命令，则 bash 不会离开，见 7.3.2 节。在任何状况下，如果 exec 失败，则交互式 Shell 不会离开。

expand_aliases

bash 会展开别名。这是交互式 Shell 的默认值。

extdebug

bash 启动除虫程序所需要的行为：

- declare -F 为每个函数名称参数显示来源文件名与行编号。
- 由 DEBUG 捕捉所执行的命令失败时，则跳过下一个命令。
- 在 Shell 函数或是 source 或 .（点号）取用的脚本内，由 DEBUG 捕捉所执行的命令失败时，则 Shell 会模拟调用 return。
- 数组变量 BASH_ARGC 被设置。每个元素都为相对应的函数或点号脚本引用，保留参数的数目。同理，BASH_ARGV 数组变量也被设置，其每个元素为传递给函数或点号脚本的其中一个参数。BASH_ARGV 函数为一个堆栈，在每次调用时，值便往前推进。因此，最后的元素即为最近函数或脚本引用的最后一个参数。
- 启用函数追踪。通过 (…) 所引用的命令替换、Shell 函数与子 Shell，都继承 DEBUG 与 RETURN 捕捉（RETURN 捕捉在 return 执行时被使用，或是一个以 .（点号）或 source 取用的脚本结束）。
- 启用错误追踪。通过 (…) 所引用的命令替换、Shell 函数与子 Shell，都继承 ERROR 捕捉。

extglob

bash 会扩展样式比对，类似 ksh88 那样。这部分请详见 14.3.3 节。

extquote

bash 允许在 ${*variable*}展开里，以双引号框住 $'...' 与 $"..."。

failglob

当样式不相符于文件名时，bash 会产生错误。

force_fignore

完成时，bash 会忽略与 FIGNORE 里字尾列表比对相符的单词，就算这样的单词是唯一可能之完成。

gnu_errfmt

bash 会以标准 GNU 格式显示错误信息。

histappend

bash 将命令附加至 HISTFILE 变量所指名的文件，而非覆盖文件。

histreedit

当历史替换失败时，如果使用 readline 程序库，则 bash 允许你重新编辑失败的替换。

histverify

使用 readline，bash 会将历史替换的结果，载入至编辑缓冲区，供进一步处理。

hostcomplete

bash 在遇到带有 @ 字符的单词时，会以 readline 执行主机名称的完成。此默认值为开启的。

huponexit

bash 会在交互式登录 Shell 离开时，传送 SIGHUP 给所有的工作。

interactive_comments

bash 视 # 为交互式 Shell 下注释的起始。此默认值为开启的。

lithist

与 cmdhist 选项合用时，bash 会存储历史里的多行命令，使用内嵌换行字符而非分号。

login_Shell

bash 会在它以登录 Shell 方式被启动时设置此选项。它无法被改变。

mailwarn

bash 在检查邮件时，如发现文件访问时间已变更，即显示"The mail in mailfile has been read"（mailfile 里的邮件已被读取）的信息。

no_empty_cmd_completion

当命令是尝试在一空行上完成时，bash 不会查找 $PATH。

nocaseglob

bash 在文件名比对时忽略大小写。

nullglob

bash 会使得不相符于任何文件的样式变成 null 字符串，而不再是表示它们自己。然后，此 null 字符串会通过更进一步的命令列处理而被删除。事实上，完全不相符的样式，会自命令列消失。

progcomp

此选项启动可程序化的完成功能。详见 *bash*(1) 手册页。其默认值为开启。

promptvars

bash 会在各种提示符号字符串的值上执行变量与参数展开。其默认值为开启。

restricted_Shell

bash 将此值设为真时，表示是以限制性 Shell 方式运作。此选项无法被改变。起始文件会查询此选项以决定其行为模式。如果想对限制性 Shell 有进一步了解，可参考 15.2 节。

shift_verbose

如果 shift 命令计数大于留下来的位置参数的数目时，则 bash 会显示信息。

sourcepath

bash 使用 $PATH 为 source 与 . 命令查找文件。默认值为启动的。如果关闭，则你必须使用完整或相对路径名称查找文件。

xpg_echo

bash 的内置 echo 会处理反斜线转义符。

14.3 共通的扩展

bash 与 ksh93 都支持大量超过 POSIX Shell 的扩展。本节要说明的是那些重叠的扩展；也就是，两个 Shell 都提供的相同功能，及以相同的方式支持。

14.3.1 select 循环

bash 与 ksh 都支持 select 循环，可轻松产生简易式选单。其语法单纯，但做的事却很多：

```
select name [in list]
do
    可以使用 $name 的语句 ...
done
```

此与一般的 for 循环具有相同的语法，只是关键字 select 不同。也就像 for 循环一样：你可以省略 in *list* 且它会默认为 "$@"；也就是被括弧起来的命令列参数的列表。

select 的行为如下：

1. 为 *list* 里的每个项目产生选单，将每个选择格式化为数字
2. 显示 PS3 的值作为提示符号，并等待用户输入一数字
3. 存储选定的选择在变量 *name* 中，以及存储选定的数字在内置变量 REPLY 里

4. 执行主体内的语句
5. 持续重复处理程序（稍后会说明如何离开）

以范例说明应较易于了解此处理过程。假设你需要知道，如何为一个分时系统正确地设置 TERM 变量，而分时系统 使用不同种类的显示终端，你没有将终端直接连到你的计算机；相反地，你的用户是通过终端服务器进行通信。虽然telnet协议可以传递TERM环境变量，但终端服务器还没有聪明到能这么做。意即tty（序列设备）号码，并不能决定终端的形态。

因此，除了在登录时提示用户终端类型之外，你没有其他选择。要作这个操作，你可以将下列代码置放在 /etc/profile（假定你有固定的一组已知终端类型）里：

```
PS3='terminal? '
select term in gl35a t2000 s531 vt99
do
    if [ -n "$term" ]
    then
        TERM=$term
        echo TERM is $TERM
        export TERM
        break
    else
        echo 'invalid.'
    fi
done
```

当你执行此代码时，会看到这样的选单：

```
1) gl35a
2) t2000
3) s531
4) vt99
terminal?
```

内置Shell变量 PS3 包含 select 使用的提示字符串，其默认值为 "#? "。为此理由，上述代码的首行将它设置为更相关的值。

select 语句是从选择列表中构建选单。如果用户输入有效数字（1至4），变量 term 被设置为相对应值；如果它是 null（用户只是按下 Enter），Shell 会再打印一次选单。

循环体里的代码会检查term是否非null。如果是，则指定$term为环境变量TERM，导出 TERM 并显示确认信息；之后break语句离开 select 循环。如果 term 是 null，则代码会显示错误信息，并再重复提示符号（但不是选单）。

break语句是离开select循环的常用方式（用户也可以输入Ctrl-D表示输入结束，以

离开select循环。这是提供予交互模式下的用户一个离开的统一方式，但对Shell程序设计师并没有什么帮助)。

我们可以让解决方案再精益求精，让选单更人性化，使得用户无须知道终端的terminfo名称。通过引文的字符字符串作为选单项目完成此任务，之后再使用case决定terminfo名称即可。新的版本呈现于例14-2。

例14-2：结合更人性化的选单项目与select

```
echo 'Select your terminal type:'
PS3='terminal? '
select term in \
    'Givalt GL35a' \
    'Tsoris T-2000' \
    'Shande 531' \
    'Vey VT99'
do
    case $REPLY in
    1) TERM=gl35a ;;
    2) TERM=t2000 ;;
    3) TERM=s531 ;;
    4) TERM=vt99 ;;
    *) echo 'invalid.' ;;
    esac
    if [[ -n $term ]]; then
        echo TERM is $TERM
        export TERM
        break
    fi
done
```

这样的代码看来与传统程序的选单子程序类似，尽管select仍提供了将选单选择转换为数字的捷径。我们让每个选单选择自已独立一行是为了可读性，不过还是得加上接续字符，以让Shell避开抱怨语法。

这里是在执行此程序时，用户将会看到的：

```
1) Givalt GL35a
2) Tsoris T-2000
3) Shande 531
4) Vey VT99
terminal?
```

这比先前代码的输出更能提供适切的信息。

进入select循环体时，$term则为4个字符串其中之一（如果用户输入无效选择，则为null)，然而内置变量REPLY则会包含用户所选定的数字。我们需要case语句，以指定正确的值给TERM，并使用REPLY的值作为case选定器。

当case语句完成后，if会检查用户的选择是否有效，这和前面的解决方案一样。如果选择有效，则TERM已被指定。所以代码只要显示确认信息，导出TERM，及离开select循环。如果非有效选择，则select循环会重复提示号，及再一次经过整个程序。

在select循环中，如果REPLY设置为null字符串，则Shell会再打印一次选单。这种状况是当用户按下Enter的时后发生。不过，你也可以直接将REPLY设为null字符串，强迫Shell再打印一次选单。

TMOUT（time out）变量会对select语句造成影响。在进入select循环之前，先将它设置为某个秒数*n*，如果在这段时间内没有任何输入数据，则select会离开。

14.3.2 扩展性 Test 工具

ksh提出扩展性test工具，以[[与]]呈现。这些是Shell的关键字，对于Shell语法是特殊的，且非命令。bash近期版本也采用此特殊工具。

[[...]]与一般test及[...]命令不同之处在于不处理单词展开与样式展开（通配字符）。意即它不需要使用引号以处理引文操作。事实上，[[...]]的内容会独自形成一个子语言，让它更易于使用。大部分的运算符都与test所使用的相同。完整列表如表14-3所示。

表 14-3：扩展 test 运算符

运算符	仅 bash 或 ksh 适用	如果为此状况，则为真
-a *file*		*file*存在。（已过时，请使用 -e）
-b *file*		*file*为区块设备文件。
-c *file*		*file*为字符设备文件。
-C *file*	ksh	*file*为连续性（contiguous）文件（绝大多数 UNIX 版本不支持）。
-d *file*		*file*为目录。
-e *file*		*file*存在。
-f *file*		*file*为一般文件。
-g *file*		*file*设置 setgid 位。
-G *file*		*file*的群组ID同于Shell下有效的群组ID。
-h *file*		*file*为符号性连接。
-k *file*		*file*设置黏着（sticky）位。
-l *file*	ksh	*file*为符号性连接（仅运作于使用 /bin/test -l 测试的符号性连接系统上）。

表 14-3：扩展 test 运算符（续）

运算符	仅 bash 或 ksh 适用	如果为此状况，则为真
-L *file*		*file* 为符号性连接。
-n *string*		*string* 为非 null。
-N *file*	bash	*file* 会被修改，因为它被读取。
-o *option*		*option* 被设置。
-O *file*		*file* 的拥有者为 Shell 的有效用户 ID。
-p *file*		*file* 为管道或命名的管道（FIFO 文件）。
-r *file*		*file* 是可读取的。
-s *file*		*file* 非空。
-S *file*		*file* 为 socket。
-t *n*		文件描述代码 *n*，指向终端。
-u *file*		*file* 设置 setuid 位。
-w *file*		*file* 是可写入的。
-x *file*		*file* 是可执行的，或是可被查找的一个目录。
-z *string*		*string* 是 null。
fileA -nt *fileB*		*fileA* 比 *fileB* 新，或 *fileB* 不存在。
fileA -ot *fileB*		*fileA* 比 *fileB* 旧，或 *fileB* 不存在。
fileA -ef *fileB*		*fileA* 与 *fileB* 指向相同文件。
string = *pattern*	ksh	*string* 相符于 *pattern*（可包含通配字符）。已过时，请使用 ==。
string = = *pattern*		*string* 相符于 *pattern*（可包含通配字符）。
string != *pattern*		*string* 不相符于 *pattern*。
stringA < *stringB*		*stringA* 在目录里的顺序先于 *stringB*。
stringA > *stringB*		*stringA* 在目录里的顺序在 *stringB* 之后。
exprA -eq *exprB*		算术表示式 *exprA* 与 *exprB* 相等。
exprA -ne *exprB*		算术表示式 *exprA* 与 *exprB* 不相等。
exprA -lt *exprB*		*exprA* 小于 *exprB*。
exprA -gt *exprB*		*exprA* 大于 *exprB*。
exprA -le *exprB*		*exprA* 小于或等于 *exprB*。
exprA -ge *exprB*		*exprA* 大于或等于 *exprB*。

运算符可以用圆括弧框起，结合 && （AND）与 || （OR）进行逻辑处理，也可使用！表示反向。当使用 /dev/fd/*n*形式的文件名时，它们会测试开放文件描述代码*n*的相对应属性。

运算符 -eq、-ne、-lt、-le、-gt，与 -ge 在 ksh93 里为已过时的用法，改用 let 命令或((...))(let 命令或((...))在 14.3.7 节里有简短的说明)。

14.3.3 扩展性样式比对

ksh88引进额外的样式比对工具让Shell的功能更强大，awk与egrep扩展正则表达式得以并驾齐驱（正则表达式的部分，详见 3.2 节）。如 extglob 选项被启用，则 bash 也支持这些运算符（它们在 ksh 里总是为启用状态）。这些额外工具的摘要整理，见表 14-4。

表 14-4：Shell 与 egrep/awk 正则表达式运算符的比较

ksh/bash	egrep/awk	含义
*(*exp*)	*exp**	存在 0 或多个 *exp*
+(*exp*)	*exp+*	存在 1 或多个 *exp*
?(*exp*)	*exp?*	存在 0 或 1 个 *exp*
@(*exp1*|*exp2*|...)	*exp1*|exp2|...	*exp1* 或 *exp2* 或…
!(*exp*)	(none)	所有不相符于 *exp* 的

Shell正则表达式与标准正则表达式的标记方式相当类似，不过它们所表示的意义不同。因为 Shell 会将表示式如 dave|fred|bob 解译为命令的管道操作，所以你必须使用 @(dave|fred|bob)这样的方式。

举例来说：

- **@(dave|fred|bob)**比对相符的有 dave、fred 或 bob。
- ***(dave|fred|bob)**意即存在 0 或多个 dave、fred 或 bob。此表示式相符的字符串，像null字符串、dave、davedave、fred、bobfred、bobbobdavefredbobfred 等等。
- **+(dave|fred|bob)**相符于上述所有字符串，null 字符串除外。
- **?(dave|fred|bob)**相符于 null 字符串，dave、fred 或 bob。
- **!(dave|fred|bob)**相符于 dave、fred 或 bob 以外的任何字符串。

我们必须再次强调：Shell 正则表达式里还是可以包含标准 Shell 通配字符。因此，Shell 通配字符?(相符于任何单一字符）等同于 egrep 或 awk 的 .（点号），且 Shell 的字符

集运算符[...]也与那些工具相同（注 1）。举例来说，表示式+([[:digit:]])相符的是数字：也就是一个或多个数字。Shell通配字符*则等同于Shell正则表达式的*(?)。你甚至可以巢状化正则表达式：+([[:digit:]]|!([[:upper:]]))表示相符于一个或多个数字，或是非大写字母。

有两个 egrep 与 awk 正则表达式运算符在 Shell 中没有等同物，它们是：

- 行开头与行结束运算符 ^ 与 $
- 单词的起始与单词的结束运算符 \< 与 \>

本质上，^ 与 $ 总是在那儿，只是样式前后带有 * 字符时会停用此功能。我们以下面的范例讲解这之间的差异：

```
$ ls                                       列出文件
biff  bob  frederick  shishkabob
$ shopt -s extglob                         启用扩展样式比对 (Bash)
$ echo @(dave|fred|bob)                    只相符于dave、fred或bob的文件
bob
$ echo *@(dave|fred|bob)*                  加入通配字符
bob frederick shishkabob                   更多文件相符
```

ksh93支持更多的样式比对运算符。但因为本节是介绍bash与ksh93之间通用的部分，所以介绍至此为止。如果想了解更多细节，可见参考书目中的*Learning the Korn Shell*(O'Reilly)。

14.3.4 括弧展开

括弧展开（Brace expansion）借自于Berkeley C Shell —— csh的功能，且两种Shell都支持它。括弧展开是让输入更轻松的方法。假设你拥有下列文件：

```
$ ls
cpp-args.c  cpp-lex.c  cpp-out.c  cpp-parse.c
```

如果你想编辑这4个文件里的其中3个，只要输入vi cpp-{args,lex,parse}.c，Shell便会展开为vi cpp-args.c cpp-lex.c cpp-parse.c。而且括弧替换还可以巢状化。例如：

```
$ echo cpp-{args,l{e,o}x,parse}.c
cpp-args.c cpp-lex.c cpp-lox.c cpp-parse.c
```

注 1： 以这点来说，乃相同于grep、sed、ed、vi等等，但是一个有名的差异是：Shell在[...]里使用！表示反向之意，而不同的工具则都使用 ^。

14.3.5 进程替换

进程替换（Process substitution）可以让用户开启多个进程数据流，再将它们喂给单一程序处理。例如：

```
awk '...' <(generate_data) <(generate_more_data)
```

（请注意此处的圆括弧也为语法的一部分，应该逐字键入。）这里，*generate_data*与*generate_more_data*表示的是用以产生数据流的任意命令，包括管道。awk程序依序处理每个数据流，不需理会数据是否来自多个来源。此部分图解说明于图 14-1.a。

进程替换也可用于输出，特别是与tee程序合用的时候，它会将其输入传送至多个输出文件以及标准输出。例如：

```
generate_data | tee >(sort | uniq > sorted_data) \
                    >(mail -s 'raw data' joe) > raw_data
```

此命令使用tee：(1) 传送数据至管道，以排序与存储数据；(2) 传送数据至mail程序，给用户joe；(3) 将原始数据重导至文件。这部分如图 14-1 所示。结合tee的进程替换，让你转义出"一个输入、一个输出"的传统 UNIX 管道思维模式，你可以将数据切分为多个输出数据流，还可以将多个输入数据流接合为一个。

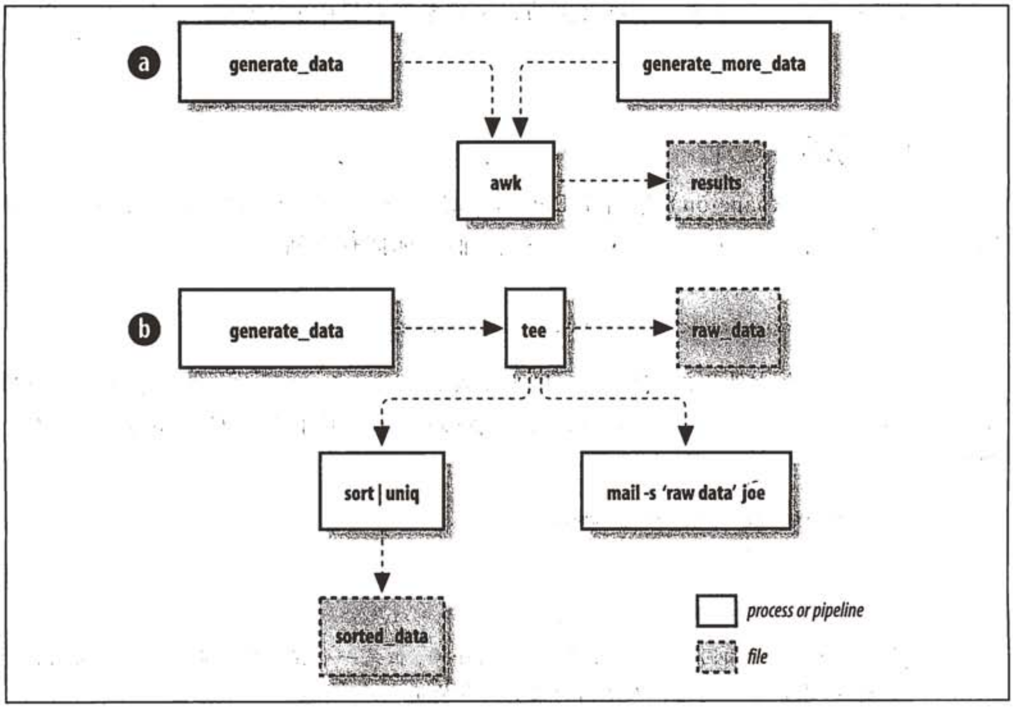


图 14-1：输入、输出数据流的进程替换

进程替换只有在支持/dev/fd/*n*特殊文件的UNIX系统下可使用，为命名访问到已开启之文件描述代码。许多现行UNIX系统，包括GNU/Linux，都支持此功能。与使用括弧展开一般，当ksh93是从原始码编译建置而成时，则它默认为启用的。bash必启用之。

14.3.6 索引式数组

ksh93与bash两者都提供索引式数组工具，虽然它很好用，但它的限制比提供类似功能的传统程序语言还多。特别是：索引式数组仅为一次元（也就是不能有数组中的数组）。索引自0起始。它们可以是任何的算术表示式：Shell会自动评估表示式产生索引。

有三种方式可以将值指定给数组里的元素。第一种为直觉式：使用标准Shell变量指定语法，将数组索引放在方括弧（[]）里，例如：

```
nicknames[2]=bob
nicknames[3]=ed
```

将值bob与ed分别置入数组nicknames的索引2与3元素中。就像正规的Shell变量一样：指定至数组元素里的值，都会视为字符字符串。

第二种将值指定至数组里的方式，是使用set语句的变体。语句如下：

```
set -A aname val1 val2 val3 ...
```

此语句建立数组*aname*（如果它原先不存在的话），然后指定*val1*至aname[0]、*val2*至aname[1]，以此类推。如你所想，这种方式比载入一组初始值至数组里方便得多。这也是ksh第一个以单一操作指定多个数组元素的方式，我们特别提它好让你在已存在的脚本中认得它。

注意： bash不支持set -A。

第三种方式（也是建议的使用方式）是使用复合指定形式：

```
aname=(val1 val2 val3)
```

欲取出数组中的值，语法为${*aname*[i]}。例如：${nicknames[2]}的值为bob。索引*i*可以是算术表示式。如果你在索引处使用*或@，则值将是以空格隔开的所有元素。省略索引($nicknames)是等同于标明索引0（${nicknames[0]}）。

现在，我们从较不一样的角度审视数组。假设，我们只指定了两个值予nicknames，也就是前面的例子。如果你输入echo "${nicknames[*]}"，会看到这样的输出：

```
bob ed
```

换句话说，nicknames[0]与nicknames[1]不存在。如果你输入：

```
nicknames[9]=pete
nicknames[31]=ralph
```

然后再 echo "${nicknames[*]}"，则输出看起来就像这样：

```
bob ed pete ralph
```

这也就是为什么先前我们说它是nicknames的索引2与3，而不说它是nicknames的第二与第三个元素。任何未指定值的数组元素都不存在，如果你试图访问其值，会得到null字符串。

你可以通过"${*aname*[@]}"（使用双引号）保留置入数组元素内的任何空白字符，而非以${*aname*[*]}。正如你会使用"$@"，而不是$*或"$*"。

两种Shell都支持${#*aname*[*]}运算符，告诉你数组里定义的元素有多少个。因此${#nicknames[*]}的值为4。请注意：[*]是必需的，因为单独的数组名称会被解译为第0个元素。意思就是，${#nicknames}等同于nicknames[0]的长度。因为nicknames[0]不存在，所以${#nicknames}的值为0，即null字符串的长度。

你可以将数组认为是采取一个整数输入变量的数学函数，及回传一相对应值（该数字里的元素）。如果你如此做，则你会了解为什么数组是"数字主控"的数据结构。由于Shell的程序设计工作多半倾向于字符字符串与文字的处理，更甚于数字，所以索引式数组工具并未广泛被使用。

尽管如此，我们还是找到索引式数组有用的地方。像我们先前在14.3.1里所提及的清理问题，用户可以在登录时选定终端类型（TERM环境变量）。例14-2使用select与case语句，呈现了更人性化的程序版本。

如我们利用select架构，将用户的数字选择存储至变量REPLY里，便能消除整个case架构。我们只需要一行代码，将所有可能的TERM存储在数组里，而且是以select选单项目所对应的顺序加以排序。之后，我们再使用$REPLY为数组作索引。代码如下：

```
termnames=(gl35a t2000 s531 vt99)
echo 'Select your terminal type:'
PS3='terminal? '
select term in \
    'Givalt GL35a' \
    'Tsoris T-2000' \
    'Shande 531' \
    'Vey VT99'
do
    if [[ -n $term ]]; then
        TERM=${termnames[REPLY-1]}
```

```
            echo "TERM is $TERM"
            export TERM
            break
        fi
    done
```

此代码设置数组termnames，所以${termnames[0]}为gl35a、${termnames[1]}为t2000等等。TERM=${termnames[REPLY-1]}主要是取代整个case架构，通过使用REPLY为数组进行索引。

这两个Shell都知道如何将数组索引中的文字解译为数值表示，好像它是被包在$((与))之间，即该变量无须前置美元符号($)。我们必须将REPLY的值减一，因为数组索引自0起始，而select选单项目编号则从1开始。

14.3.7 各类扩展

这里又是另一串长长的列表，这次的内容是bash与ksh93所支持的POSIX Shell小型扩展：

附加的波浪号展开

POSIX将文字~定义为等同于$HOME与~*user* —— *user*的根目录。这两个Shell都允许使用~+作为$PWD（当前工作目录）的缩写，使用~-作为$OLDPWD（前一个工作目录）的缩写。

算术命令

POSIX定义$((...))标记作为算术展开，但不提供任何其他算术操作的机制。不过，两种Shell都支持两种直接处理算术的标记，而非展开：

```
let "x = 5 + y"                    let 命令，需以引号框起
((x = 5 + y))                      未前置 $，自动的用双圆括弧引起来
```

我们并不清楚为什么POSIX仅将算术展开标准化，可能是由于你可以使用：(do-nothing）命令与算术展开达到相同效果：

```
: $((x = 5 + y))                   几乎与 let 或 ((…)) 一样
x=$((5 + y))                       类似，但 = 前后都不可置放任何空格
```

有个不同之处，便是let与((...))都有离开状态：0为真（true）值，而1为伪（false）值。这一点，让你能在if与while语句里使用它们：

```
while ((x != 42))
do
    ...  任何东西  ...
done
```

算术的for循环

两个Shell都支持算术的for循环，它和awk、C与C++里的for循环很相似。看起来就像这样：

```
for ((init; condition; increment))
do
    循环体
done
```

这里面的*init*、*condition*，与*increment*任一个，都可为Shell的算术表示式，正如同它出现在`$((...))`里那样。在`for`循环里使用`((...))`语法相似于算术评估语法。

当你需要以固定次数执行任务时，可以使用算术的`for`循环：

```
for ((i = 1; i <= limit; i += 1))
do
    任何需要做的事情
done
```

附加的算术运算符

POSIX定义了可置于算术展开`$((...))`里的运算符列表。两种Shell都支持额外的运算符，提供与C完整的兼容性。特别是两个Shell都允许使用`++`与`--`分别执行加减一的操作，且前置或置于结尾的形式都允许（根据POSIX的定义：`++`与`--`都是选择性的）。除此之外，两个Shell都支持逗点运算符，它可以让你在单一表示式里执行多个运算。而且，更甚于C的是：两个Shell都接受`**`作取幂操作。运算符完整列表如表14-5所示。

`case` *语句的可选用圆括弧比对*

命令替换的`$(...)`语法（见7.6节）已由POSIX标准化。它是由`ksh88`引进，但`bash`也支持之。`ksh88`在`$(...)`里处理`case`语句时会有问题，尤其是关闭用的右圆括弧被用于每个`case`样式时，都会终止整个命令替换。要解决这个问题，在命令替换下，`ksh88`必须将`case`样式包括在比对的圆括弧里：

```
some command $( ...
    case $var in
    ( foo | bar )     some other command ;;
    ( stuff | junk )  something else again ;;
    esac
    ... )
```

`ksh93`、`bash`与POSIX都可在`case`选择器里使用一个可选用的开启圆括弧，不过并不是一定要这么做（因此，`ksh93`较`ksh88`聪明，后者要求开启圆括弧必须置于`$(...)`内）。

`trap -p` *显示捕捉*

根据POSIX说明，原先的`trap`命令会显示Shell的捕捉状态，而且是采取Shell日后可重新读取的形式，以回复相同的捕捉。两种Shell都允许使用`trap -p`显示捕捉。

使用 <<< 的即席字符串

以 echo 产生单行输入供进一步处理，是相当常见的用法。

例如：

```
echo $myvar1 $myvar2 | tr ... | ...
```

两种 Shell 都支持我们所谓的即席字符串（here strings）标记方式，这是取自 rc Shell（注 2）的 UNIX 版本。即席字符串使用 <<< 再接上字符串。字符串即成为相关联命令的标准输入，辅以 Shell 自动提供最后的换行符号：

```
tr ... <<< "$myvar1 $myvar2" | ...
```

这么做可以不用建立额外的进程，且标记的方式也更清楚。

扩展性字符串标记

bash 与 ksh93 都支持特殊字符串标记，可了解一般类 C（或类 echo）的转义符。此标记包括在单引号框起的字符串，前置一个 $。这类字符串的行为模式有点像一般单引号框起来的字符串，不过 Shell 会解译字符串里的转义符。例如：

```
$ echo $'A\tB'                A、定位字符、B
A       B
$ echo $'A\nB'                A、换行字符、B
A
B
```

表 14-5 列出的是 bash 与 ksh93 都支持的算术运算符。

表 14-5：bash 与 ksh93 的算术运算符

运算符	意含	相关性
++ --	加一或减一，可前置或置于结尾	由左至右
+ - ! ~	单元加号或减号，逻辑与 bitwise 否定用法	由右至左
**	取幂[注]	由右至左
* / %	乘、除及余数	由左至右
+ -	加与减	由左至右
<< >>	向左与向右位移一个位	由左至右
< <= > >=	比较	由左至右
= = !=	等于与不等	由左至右
&	Bitwise 的 AND	由左至右
^	Bitwise 的 Exclusive OR	由左至右

注 2： 见 *http://www.star.le.ac.uk/~tjg/rc/*。

表 14-5：bash 与 ksh93 的算术运算符（续）

运算符	意含	相关性
|	Bitwise 的 OR	由左至右
&&	逻辑的 AND（捷径式）	由左至右
||	逻辑的 OR（捷径式）	由左至右
?:	条件表示式	由右至左
= += -= *= /= %= &= ^= <<= >>= |=	指定运算符	由右至左
,	连续性评估	由左至右

注：ksh93m 与更新版本使用。在 bash 3.1 之前的版本里，** 为左相关联的。它是自 3.1 版后才变成右相关联处理。C 语言里没有 ** 运算符。

圆括弧可用于群组化子表示式。其算术表示式的语法（类 C）支持关联式运算符：1 为真、0 为伪。

例如：`$((3 > 2))` 的值为 1；`$(( (3 > 2) || (4 <= 1) ))` 也是值为 1：因为两个子表示式里至少有一个为真即可。

14.4 下载信息

本节将简略说明到哪里去寻找 bash 与 ksh93 的源代码，还有如何以源代码建置 Shell。我们假定你的系统里已经有 C 编译器，还有 make 程序。

14.4.1 bash

bash 可自 Free Software Foundation GNU Project 的 FTP 服务器取得。此书编写的同时，最新的现行版本为 3.0。你可以使用 wget（如果你有的话）直接取得发布的 tar 文件：

```
$ wget ftp://ftp.gnu.org/gnu/bash/bash-3.0.tar.gz
--17:49:21--  ftp://ftp.gnu.org/gnu/bash/bash-3.0.tar.gz
           => `bash-3.0.tar.gz'
...
```

另外，你也可以使用传统匿名 FTP 的方式取得：

```
$ ftp ftp.gnu.org                                   FTP 至该服务器
Connected to ftp.gnu.org (199.232.41.7).
220 GNU FTP server ready.
Name (ftp.gnu.org:tolstoy): anonymous               匿名登录
230 Login successful.
230-Due to U.S. Export Regulations, all cryptographic software on this
```

```
230-site is subject to the following legal notice:
...
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> cd /gnu/bash                                   切换至 bash 目录
250 Directory successfully changed.
ftp> binary                                         确认为二进制模式
200 Switching to Binary mode.
ftp> hash                                           显示 # 标记
Hash mark printing on (1024 bytes/hash mark).
ftp> get bash-3.0.tar.gz                            取出文件
local: bash-3.0.tar.gz remote: bash-3.0.tar.gz
227 Entering Passive Mode (199,232,41,7,149,247)
150 Opening BINARY mode data connection for bash-3.0.tar.gz (2418293
bytes).
###########################################################################
###########################################################################
...
226 File send OK.
2418293 bytes received in 35.9 secs (66 Kbytes/sec)
ftp> quit                                        大功告成
221 Goodbye.
```

除了bash发布包本身以外，你应该也要取回任何的修补文件（patch）。以3.0版的bash而言，它的修补文件 —— 修改源代码应修正的部分 —— 必须自不同的地点取得。你可以在*ftp://ftp.cwru.edu/pub/bash/bash-3.0-patches/*下找到，取出所有修补文件后，将之置于临时目录，方式如下：

```
$ mkdir /tmp/p              建立临时性目录
$ cd /tmp/p                 切换过去
$ for i in 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16
> do wget ftp://ftp.cwru.edu/pub/bash/bash-3.0-patches/bash30-0$i
> done                      取出所有修补文件
...  省略许多的输出  ...
```

写这本书时，共有16个修补文件，有可能还有更多更新的修补文件，得视bash版本而定。

至此，你已准备好解开发布文件与套用修补文件了。首先，解开源代码：

```
$ gzip -d < bash-3.0.tar.gz | tar -xpvzf -     解压缩与解开打包文件
bash-3.0/
bash-3.0/CWRU/
bash-3.0/CWRU/misc/
bash-3.0/CWRU/misc/open-files.c
bash-3.0/CWRU/misc/sigs.c
...  省略许多输出  ...
```

然后，套用修补文件：

```
$ cd bash-3.0                                             切换至原始码的目录
$ for i in /tmp/p/*                                       套用所有修补文件
> do patch -p0 --verbose --backup < $i
> done
...  省略许多的输出  ...
$ find . -name '*.rej'                                    检查是否失败
$ find . -name '*.orig' -print | xargs rm                 清空
```

修补文件的引用一如上述，是使用GNU版本的patch。请注意有些商用UNIX系统提供的是较旧的版本。在套用修补文件之后，我们会寻找 .rej（reject）文件，看看是否有修补失败的记录，在这里没有，所以一切运作正常。接下来，我们会删除.orig(original)文件，再以下面的操作建置 bash：

```
$ ./configure && make && make check                       配置、建置、测试
checking build system type... i686-pc-linux-gnu
checking host system type... i686-pc-linux-gnu
...  省略许多输出  ...
```

如果所有的检测通过（也应该是如此），那就表示大功告成了！你可以使用 make install 安装最新建置完成的 bash 可执行文件（可能得切换为 root 的身份才能做这件事）。

14.4.2 ksh93

ksh93 的源代码可至 AT&T Research 网站下载，URL 为 *http://www.research.att.com/sw/download*。ksh93的建置是相当直觉的，但处理方式较bash多了许多手动操作的部分。我们所呈现的是 2004 年 2 月建置 ksh93p 的步骤，其流程与现行版本相似。在此我们选择仅建置 Korn Shell，不过你应该比较希望下载与建置整个“AST Open”包，因为它提供了很完整的工具组。

1. 自网站下载包 INIT.2004-02-29.tgz 与 ast-ksh.2004-02-29.tgz。将之置于某个空目录下，以供将来建置软件所用。

2. 建立 lib/package/tgz 目录，并将这两个文件移过去：

   ```
   $ mkdir -p lib/package/tgz
   $ mv *.tgz lib/package/tgz
   ```

3. 手动解开 INIT 包：

   ```
   $ gzip -d < lib/package/tgz/INIT.2004-02-29.tgz | tar -xvf -
   ...  省略许多输出  ...
   ```

4. 借由读取哪个包可用，使用 AT&T 的工具开始建置流程：

   ```
   $ bin/package read
   package: update /home/tolstoy/ksh93/bin/execrate
   ...  省略许多输出  ...
   ```

5. 再使用 AT&T 工具，开始编译：

```
$ bin/package make
package: initialize the /home/tolstoy/ksh93/arch/linux.i386 view
...  省略许多输出 ...
```

此步骤会花费一些时间，视你系统与编译器的速度而定。

6. 新建置完成的 ksh93 二进制文件，位于 arch/*ARCH*/bin/ksh 下，其中，*ARCH* 表示的是你建置 ksh93 那台机器的架构。以 x86 GNU/Linux 系统而言，即为 linux.i386。像这样：

```
$ arch/linux.i386/bin/ksh            执行新建置的 ksh93
$ echo ${.sh.version}                显示版本
Version M 1993-12-28 p
```

7. 或许你会想要将新建置完成的 Korn Shell 移至你路径内的目录里，例如这样的个人 bin 目录：

```
$ cp arch/linux.i386/bin/ksh $HOME/bin/ksh93
```

好了，可以开始使用了！

14.5 其他扩展的 Bourne 式 Shell

另外还有两个也是相当受欢迎的 Shell：

Public Domain Korn Shell

许多开放源码的类 UNIX 系统，像是 GNU/Linux，都随附 Public Domain Korn Shell，pdksh。pdksh 源代码可自 *http://web.cs.mun.ca/~michael/pdksh/* 取得，其附有命令供用户建置与安装在各种 UNIX 平台上。

pdksh 原由 Eric Gisin 所编写，它将 pdksh 的基础建置在 Charles Forsyth 的 Version 7 Bourne Shell 之公众领域版本上。有许多部分兼容于 1988 Korn Shell 与 POSIX，另带有部分自有的扩展。

Z-Shell

zsh 是一套强而有力的交互式 Shell 与脚本语言，能做到 ksh、bash 与 tcsh 所能完成的很多任务，也有许多特有的功能。zsh 拥有 ksh88 大部分的功能但 ksh93 的倒是极少。它完全可以自由取得，并应被编译与执行于任何现代的 UNIX 版本上。置于其他操作系统上也可行，其官方网站为 *http://www.zsh.org/*。

这些 Shell 在 *Learning the Korn Shell*（O'Reilly）里都有更详尽的叙述，书籍信息详见参考书目。

14.6 Shell版本

这些对于扩展性Shell的探讨提醒我们，偶而查查各种不同Shell的版本信息会是很方便的。查询方法如下：

```
$ bash --version                                 bash
GNU bash, version 3.00.16(1)-release (i686-pc-linux-gnu)
...

$ ksh --version                                  仅最近的 ksh93 适用
  version         sh (AT&T Labs Research) 1993-12-28 p

$ ksh                                            旧版的 ksh
$ ^V                                             键入 ^V
$ Version 11/16/88f                              ksh 显示其版本

$ echo 'echo $KSH_VERSION' | pdksh               pdksh
@(#)PD KSH v5.2.14 99/07/13.2

$ echo 'echo $ZSH_VERSION' | zsh                 zsh
4.1.1
```

此处并未提供取得/bin/sh版本编号的方式。这没什么好惊讶的，大部分商用UNIX系统上真正的Bourne Shell，都系出System V Release 3（1987）或Release 4（1989）Bourne Shell，之后改变很少，甚至可以说是没有任何变动。厂商希望采用某个Korn Shell版本，以提供POSIX兼容的Shell。

14.7 Shell初始化与终止

为支持用户客户化，Shell会在启动、终止时，读取某些特定文件。每个Shell都有不同的惯例模式，所以我们以独立的小节分别讨论它们。

如果你编写的Shell脚本希望能被其他Shell使用，你就不应该依赖启动时的定制功能。我们在本书所开发的所有Shell脚本都会设置它们自己的环境（例如$PATH的值），让任何人都能执行它们。

Shell的行为模式端视它是否为登录Shell（*login Shell*）而定。当你坐在终端前，在计算机的提示符号下输入你的username与密码时，便是正在取得登录Shell。相同地，当你使用ssh *hostname*时，也是取得一个登录Shell。然而，如果你指定名称执行Shell，或直接在脚本首行#!下指定的命令解释器执行，或建立一个新的工作站终端窗口，或在远端Shell中执行命令时，例如ssh *hostname command*，则该Shell都不是登录Shell。

Shell是检查$0的值决定它是否为登录Shell，如果该值以连字号起始，即为登录Shell，否则就不是。你可以使用下列方式，得知你现在是否在登录Shell下：

```
$ echo $0                                       显示 Shell 名称
-ksh                                            是的，这是登录 Shell
```

连字号并不是暗示有个文件名叫 /bin/-ksh。它只是表示父进程设置第0个参数，当它执行 exec()系统调用以启动 Shell 时使用。

如果你日常处理都只有一种Shell，那么后面几个小节所叙述的初始与终结文件对你来说就不是问题：你只要设置好一次，定制为你要的模式，确认运作无误，就可以很久不去动它。但如果你得处理数种Shell，可能就必须更审慎考虑你的定制方式，避免重复与维护的头疼问题。.（点号）与 test 命令都是不可或缺的好工具，你可以将它们用在定制的脚本里，读取可让 Bourne 家族 Shell 接受的一组小心写入之文件，也可以将它们用于所有你必须访问的主机上。系统管理者也需要将全面系统地定制脚本置于 /etc 下，让所有用户都可使用。

14.7.1 Bourne Shell（sh）启动

当它为登录 Shell 时，Bourne Shell —— sh 相当于：

```
test -r /etc/profile && . /etc/profile            尝试读取 /etc/profile
test -r $HOME/.profile && . $HOME/.profile        尝试读取 $HOME/.profile
```

即它会读取两个与当前 Shell 相关的启动文件，但不强求它们一定要有一个存在。需特别留意的是，根目录文件为一个点号文件，而在 /etc 下系统面的则否。

系统 Shell 启动文件的建置，是由本地端管理，看起来像这样：

```
$ cat /etc/profile                       显示系统 Shell 启动文件
PATH=/usr/local/bin:$PATH                将 /usr/local/bin 加入至系统路径起始处
export PATH                              导出让子进程知道
umask 022                                删除群组与其他人的写入权限
```

传统的 $HOME/.profile文件可以修改本地端系统的默认登录环境，使用的命令如下：

```
$ cat $HOME/.profile                     显示个人的 Shell 启动文件
PATH=$PATH:$HOME/bin                     将个人 bin 目录加入至系统路径的结尾处
export PATH                              导出让子进程知道
alias rm='rm -i'                         文件删除时要求确认
umask 077                                删除群组与其他人的所有访问
```

子 Shell 接连建立后会继承父 Shell 的环境字符串，包括 PATH。它也继承当前工作目录与当前文件权限掩码，两者都记录于内核里进程专有的数据中。然而，它不会继承其他的定制，像 alias 所设置的命令缩写用法或是未导出的变量。

当 Shell 不是一个登录 Shell 时，Bourne Shell 并未提供自动读取启动文件功能，所以别名的使用会受限。由于远端命令执行也不会建立登录Shell，所以你也不能指望 PATH 会

设置为你惯用的值：它可能就只是 /bin:/usr/bin这么简单的设置。我们在8.2节里的 build-all 脚本里已处理过这种状况。

在离开时，Bourne Shell 不会读取标准的终结文件，但你可以设置一个捕捉，让它这么做(我们在13.3.2已说明捕捉的细节了)。例如，如果你将这个语句放在$HOME/.profile 里：

```
trap '. $HOME/.logout' EXIT
```

然后$HOME/.logout脚本便能执行所有你要的清除操作，像是以clear命令清扫屏幕。然而，因为只能为任何给定的一个信号指定一个捕捉，所以如果是它在稍后的通信期里被覆盖时，便会失去这里指定的捕捉：没有方法可保证终结的脚本一定会被执行。对于非登录 (nonlogin) Shell，每个需要离开处理的脚本或通信期，都必须明确地设置 EXIT 捕捉，但一样不保证在离开时必定会生效。

这些限制、缺乏命令历史 (history) 的支持 (注 3)，以及一些较旧的实例、工作控制，都让 Bourne Shell 不为多数交互模式下的用户所青睐。在大部分商用 UNIX 系统上，倾向于只让 root 或其他系统管理者身份的账号，在短暂的通信期下交互式地使用它。尽管如此，Bourne Shell 还是可移植性 Shell 脚本期待的选择。

14.7.2 Korn Shell 启动

当 Korn Shell - ksh 启动为登录 Shell时，它会像 Bourne Shell 一样读取 /etc/profile 与 $HOME/.profile —— 如果这两个文件存在且可读取的话。

当 ksh93 启动为交互式 Shell (登录或非登录)，它会作这样的操作：

```
test -n "$ENV" && eval . "$ENV"                    试着读取 $ENV
```

ksh88 无条件处理 $ENV，针对所有的 Shell。

eval 命令在7.8节里已说明。现在，我们已经知道它先评估它的参数，所以任何在那里的变量都会被展开，然后以命令的方式执行结果字符串。效果便是在当前 Shell 下，读取并执行 ENV 里指名的文件。PATH 目录并不会用来查找这里的文件，所以 ENV 应标明一个绝对路径。

ENV 的功能解决了 Bourne Shell 为子 Shell 通信期设置私有别名的问题。然而，它并没解决非登录远端通信期的定制问题：它们的 Shell 永远不会读取任何的初始化文件。

注 3： 很多系统上的 /bin/sh 仅为 bash 的连接，在这种情况下，命令历史是可以使用的。不过，原始的 Unix Bourne Shell 并不支持命令历史功能。

非交互式的 ksh93 就像 Bourne Shell 一样，不会读取任何的启动脚本，也不会在离开之前读取任何的终结脚本，除非你自已下达 trap 命令（正如之前所言，一个非交互式 ksh88 会在启动时读取及执行 $ENV 文件）。

14.7.3 Bourne-Again Shell 启动与终结

虽然GNU bash时常是以自已的方式被作为登录Shell，但当它以sh的名称引用时，也可以模拟为 Bourne Shell。而它接下来的启动行为就如 14.7.1 节里所描述一般，在此种状况下本节大部分的内容都不能套用。在 GNU/Linux 系统下，/bin/sh 必为 /bin/bash 的符号性连接。

警告： bash 的模拟 Bourne Shell 并不完美，因为当 bash 被引用为 sh 时，会隐藏其诸多扩展功能的一部分。我们偶然发现软件包里的Shell脚本是开发在由 /bin/sh执行的GNU/Linux环境下，但并未在真正的 Bourne Shell 环境下测试，后者环境很可能因为用的是扩展性功能而导致它们失败。

当 bash 为登录 Shell 时，启动时它的操作相当于：

```
test -r /etc/profile && . /etc/profile              试着读取 /etc/profile
if test -r $HOME/.bash_profile ; then               尝试三种可能性
    . $HOME/.bash_profile
elif test -r $HOME/.bash_login ; then
    . $HOME/.bash_login
elif test -r $HOME/.profile ; then
    . $HOME/.profile
fi
```

系统面的文件则同于Bourne Shell的，只不过在 $HOME 内的查找顺序允许你将bash特定的初始文件放进这两个文件其中之一。否则，bash会回去读取你个人的Bourne-Shell启动文件。

离开时，一个 bash 登录 Shell 会做：

```
test -r $HOME/.bash_logout && . $HOME/.bash_logout    试着读取终结脚本
```

不同于Bourne Shell的是，bash在交互式非登录Shell下启动时会读取初始文件，步骤相当于：

```
test -r $HOME/.bashrc && . $HOME/.bashrc              试着读取 $HOME/.bashrc
```

此状况下，就不会读取登录 Shell 的启动文件。

当bash使用在非交互模式时，不会读取 .bashrc文件或登录Shell启动文件，它改为读取定义于BASH_ENV变量中的文件，像这样：

```
test -r "$BASH_ENV" && eval . "$BASH_ENV"          试着读取$BASH_ENV
```

以ksh来说，PATH目录并不是用来查找这个文件的。

请留意它们的差异：Korn Shell的ENV变量仅用于非登录的交互式Shell，而bash的BASH_ENV则仅用于非交互式Shell。

要理清启动文件处理的顺序，我们使用echo命令进行测试。登录通信期看起来就像这样：

```
$ login                                         起始一个新的登录session
login: bones
Password:                                       抑制密码输出
DEBUG: This is /etc/profile
DEBUG: This is /home/bones/.bash_profile
$ exit                                          终结通信期
logout
DEBUG: This is /home/bones/.bash_logout
```

交互式通信期仅引用一个文件：

```
$ bash                                          开始交互式通信期
DEBUG: This is /home/bones/.bashrc
$ exit                                          终结通信期
exit
```

非交互式通信期通常不会引用任何文件：

```
$ echo pwd | bash                               于bash之下执行命令
/home/bones
```

然而，如果BASH_ENV值指向一个起始文件时，便会这么做：

```
$ echo pwd | BASH_ENV=$HOME/.bashenv bash       在bash下执行命令
DEBUG: This is /home/bones/.bashenv
/home/bones
```

14.7.4 Z-Shell起始与终结

Z-Shell —— zsh可仿为Bourne Shell或是Korn Shell。在sh或ksh名称下被引用，或是以字符s或k开头的任何名称被引用时，可选择性地前置单一r（restricted；限制性），它就会拥有与那些Shell相同的启动行为，且本节的其他部分不能套用（模仿ksh时，它会遵循总是处理$ENV文件的ksh88行为模式）。

Z-Shell拥有最复杂，也最弹性的定制处理。每次Z-Shell启动，无论它是否为登录Shell、交互式Shell，或是非交互式Shell，都会试着读取两个初始文件，像这样：

```
test -r /etc/zshenv && . /etc/zshenv                    读取系统面的脚本
if test -n "$ZDOTDIR" && test -r $ZDOTDIR/.zshenv ; then
    . $ZDOTDIR/.zshenv                                  读取此文件
elif test -r $HOME/.zshenv ; then
    . $HOME/.zshenv                                     或此文件
fi
```

ZDOTDIR变量是防止zsh自动读取用户根目录下的启动文件的系统管理手段，相对地，它会强制读取在管理控制下位于其他地方的文件。如果需使用此变量，则它会被设置在 /etc/zshenv 里，所以你可以到那儿看看你的系统处理方式。

假设ZDOTDIR未设置，最好的位置便是将它置于个人定制的地方，其中你希望在每一个Z-Shell 通信期中都能生效。$HOME/.zshenv是可以影响所有 Z-Shell 通信期的文件。

如为登录 Shell，接下来它会执行相当于下列的命令，读取两个启动 profile：

```
test -r /etc/zprofile && . /etc/zprofile                 读取系统面的脚本
if test -n "$ZDOTDIR" && test -r $ZDOTDIR/.zprofile ; then
    . $ZDOTDIR/.zprofile                                 读取此文件
elif test -r $HOME/.zprofile ; then
    . $HOME/.zprofile                                    或此文件
fi
```

如果为登录 Shell 或交互式 Shell，接下来会试图读取两个启动文件，像这样：

```
test -r /etc/zshrc && . /etc/zshrc                      读取系统面脚本
if test -n "$ZDOTDIR" && test -r $ZDOTDIR/.zshrc ; then
    . $ZDOTDIR/.zshrc                                   读取此文件
elif test -r $HOME/.zshrc ; then
    . $HOME/.zshrc                                      或此文件
fi
```

最后，如果为登录 Shell，它还会试着读取两个登录脚本，像这样：

```
test -r /etc/zlogin && . /etc/zlogin                    读取系统面脚本
if test -n "$ZDOTDIR" && test -r $ZDOTDIR/.zlogin ; then
    . $ZDOTDIR/.zlogin                                  读取此文件
elif test -r $HOME/.zlogin ; then
    . $HOME/.zlogin                                     或此文件
fi
```

zsh离开时，如果为登录Shell，它不会因为由exec执行的其他进程而被终结，而是借由读取两个终结脚本而结束。依序为一个用户的，一个系统的：

```
if test -n "$ZDOTDIR" && test -r $ZDOTDIR/.zlogout ; then    读取此文件
    . $ZDOTDIR/.zlogout
```

```
elif test -r $HOME/.zlogout ; then            或此文件
    . $HOME/.zlogout
fi
test -r /etc/zlogout && . /etc/zlogout        读取系统面脚本
```

Z-Shell初始与终结的处理程序相当复杂。为了更了解它做了什么，我们将每个文件搭配 echo 命令，且将 ZDOTDIR 停留在未设置状态。这么一来，只有在 /etc 与 $HOME 下的文件才会被寻找。登录通信期看起来就会像这样：

```
$ login                                    起始新的登录通信期
login: zabriski
Password:                                  抑制密码显示
DEBUG: This is /etc/zshenv
DEBUG: This is /home/zabriski/.zshenv
DEBUG: This is /etc/zprofile
DEBUG: This is /home/zabriski/.zprofile
DEBUG: This is /etc/zshrc
DEBUG: This is /home/zabriski/.zshrc
DEBUG: This is /etc/zlogin
DEBUG: This is /home/zabriski/.zlogin
$ exit                                     终结通信期
DEBUG: This is /home/zabriski/.zlogout
DEBUG: This is /etc/zlogout
```

交互式通信期引用较少的文件：

```
$ zsh                                      开始一个新的交互式通信期
DEBUG: This is /etc/zshenv
DEBUG: This is /home/zabriski/.zshenv
DEBUG: This is /etc/zshrc
DEBUG: This is /home/zabriski/.zshrc
$ exit                                     终结通信期
                                           静默：未读取任何终结文件
```

非交互式通信期仅使用两个文件：

```
$ echo pwd | zsh                           在 zsh 下执行命令
DEBUG: This is /etc/zshenv
DEBUG: This is /home/zabriski/.zshenv
/home/zabriski
```

14.8 小结

POSIX 标准提升了可移植式 Shell 脚本的可能性。如果你在该定义下做事，那么要完成一个可移植式脚本是有机会成功的。不过真实世界往往更复杂。虽然 bash 与 ksh93 都提供比 POSIX 还多很多的扩展，但它们也并非百分之百彼此兼容。我们列出了一长串"迷思（Gotchas）"列表，将它们挑出来，甚至还包括了像 set 选项或是存储 Shell 的完整状态等领域，值得大家注意。

shopt命令可用以控制bash的行为。我们特别建议你：在交互模式下启用extglob选项。

bash与ksh93共享许多共通的扩展——在Shell程序化时相当好用：select循环、扩展性测试工具[[...]]、扩展性样式比对、括弧展开、进程替换及索引式数组。我们也提到许多其他小型，但很有用的扩展。算术for循环与((...))算术命令可能是这些里面最有名的了。

bash与ksh93的源代码可自Internet下载取得，我们还介绍了这两个Shell的建置方式。除此之外，还有两个广受欢迎的扩展性Bourne式Shell：pdksh与zsh，我们也做了简短的介绍。

我们让你知道如何确认你所执行的Shell版本。这在你需要知道正在使用的程序是哪个版本时会用得到。

最后，不同的Bourne Shell语言实例，在起始与终结时都有不同的定制功能与文件。Shell脚本倾向于一般的使用，不要依赖任何个别用户所设置的功能或变量，反而应该由他们自行处理所有必需的初始化操作。

第 15 章

安全的 Shell 脚本：起点

UNIX 的安全性一向是恶名在外。几乎从每个角度看，UNIX 系统都有或多或少的安全性争议，不过这些大部分都是系统管理者该担心的。

本章，我们会先列出一长串“诀窍”，提醒你编写 Shell 脚本应该注意的地方，以避开安全性问题。再者，我们会介绍限制性 Shell（restricted Shell），这是对用户环境加以限制的 Shell。接下来，我们还会提到特洛伊木马（Trojan horse），并说明为什么应该要避开它。最后，将探讨setuid的Shell脚本，包括Korn Shell的特权模式(privileged mode)。

注意： 本书主题并非探讨 UNIX 系统安全性。本章进行的讨论也只是冰山一角，而 UNIX 系统的安全性除了 Shell 的设置之外，还有无数的层面必须关切。

如果你想了解更多 UNIX 的安全性问题，我们建议你看《Internet Security》（O'Reilly）这本书（见参考书目）。

15.1 安全性 Shell 脚本提示

下面的提示，有助于你编写一个更安全的 Shell 脚本，感谢 Purdue University 的 Center for Education and Research in Information Assurance and Security学者Professor Eugene (Gene) Spafford（注 1）所提供的信息：

勿将当前目录（点号）置于 PATH 下

可执行程序应该只能放在标准的系统目录下，将当前目录（点号）放在 PATH 里，无疑是打开特洛伊木马（Trojan Horse）的大门，详见 15.3 节。

注 1： 见 *http://www.cerias.purdue.edu/*。

为bin目录设置保护

确认$PATH下的每一个目录都只有它的拥有者可以写入，其余任何人都不能。同样的道理也应应用于bin目录里的所有程序。

写程序前，先想清楚

花点时间想想，你想要做的是什么、该如何实行。不要一开始就在文字编辑器上直接写，而且，在它真的开始运作前，要不断地设法测试。错误与失败的优雅处理，也应该设计在程序里。

应对所有输入参数检查其有效性

如果你期待的是数字，那就验证你拿到的是数字。检查该数字是否在正确的范围内。同样的检查也应该出现在其他类型的数据上，Shell的模式匹配工具可以将这个工作处理得很好。

对所有可返回错误的命令，检查错误处理代码

不在你预期内的失败情况，很可能是有问题的强迫失败，导致脚本出现不当的行为。例如，如果参数为NFS加载磁盘或面向字符的设备文件时，即便是以root的身份执行，也可能导致有些命令失败。

不要信任传进来的环境变量

如果它们被接下来的命令（例如TZ、PATH、IFS等）使用时，请检查并重设为已知的值。ksh93会在启动时自动重设IFS为它的默认值，无论当时环境为何，但其他许多Shell就不会这么做了。无论在什么情况下，最好的方式就是明确地设置PATH只包含系统bin目录，并设置IFS为“空格－定位字符－换行字符”(space-tab-newline)。

从已知的地方开始

在脚本开始时，确切cd到已知目录，这么一来，接下来的任何相对路径名称才能指到已知位置。请确认cd操作成功：

```
cd app-dir || exit 1
```

在命令上使用完整路径

这么做你才能知道自己使用的是哪个版本，无须理会$PATH设置。

使用syslog(8)保留审计跟踪

记录引用的日期与时间、username等，参见*logger*(1)的使用手册。如果你没有logger，可建立一个函数保留日志文件：

```
logger(){
    printf "%s\n" "$*" >> /var/adm/logsysfile
}
logger "Run by user " $(id -un) "($USER) at " $(/bin/date)
```

当使用该输入时，一定将用户输入引用起来

例如："$1"与"$*"，这么做可以防止居心不良的用户输入作超出范围的计算与执行。

勿在用户输入上使用 eval

甚至在引用用户输入之后，也不要使用 `eval` 将它交给 Shell 再处理。如果用户读了你的脚本，发现你使用 `eval`，就能很轻松地利用这个脚本进行任何破坏。

引用通配字符展开的结果

你可以将空格、分号、反斜杠等放在文件名里，让棘手的事情交给系统管理员处理。如果管理的脚本未引用文件名参数，此脚本将会造成系统出问题。

检查用户输入是否有 meta 字符

如果使用 `eval` 或 `$(...)` 里的输入，请检查是否有像 `$` 或 `` ` ``（旧式命令替换）这类的 meta 字符。

检测你的代码，并小心谨慎阅读它

寻找是否有可被利用的漏洞与错误。把所有坏心眼的想法都考虑进去，小心研究你的代码，试着找出破坏它的方式，再修正你发现的所有问题。

留意竞争条件（race condition）

攻击者是不是可以在你脚本里的任两个命令之间执行任意命令，这对安全性是否有危害？如果是，换个方式处理你的脚本吧！

对符号性连接心存怀疑

在 `chmod` 文件或是编辑文件时，检查它是否真的是一个文件，而非连接到某个关键性系统文件的符号性连接（利用`[ -L file ]`或`[ -h file ]`检测*file*是否为一符号性连接）。

找其他人重新检查你的程序，看看是否有问题

通常另一双眼睛才能找出原作者在程序设计上陷入的盲点。

尽可能用 setgid 而不要用 setuid

这些术语在本章稍后有探讨。简而言之，使用setgid能将损害范围限制在某个组内。

使用新的用户而不是 root

如果你必须使用 setuid 访问一组文件，请考虑建立一个新的用户，非 `root` 的用户做这件事并设置 setuid 给它。

尽可能限制使用 setuid 的代码

尽可能让setuid代码减到最少。将它移到一个分开的程序，然后在大型脚本里有需要时才引用它。无论如何，请做好代码防护，好像脚本可以被任何人于任何地方引用那样！

bash维护工程师Chet Ramey提供了下列代码的开场白，给那些需要更多安全性的Shell脚本使用：

```
# Reset IFS. Even though ksh doesn't import IFS from the environment,
# $ENV could set it.  This uses special bash and ksh93 notation,
# not in POSIX.
IFS=$' \t\n'

# Make sure unalias is not a function, since it's a regular built-in.
# unset is a special built-in, so it will be found before functions.
unset -f unalias

# Unset all aliases and quote unalias so it's not alias-expanded.
\unalias -a

# Make sure command is not a function, since it's a regular built-in.
# unset is a special built-in, so it will be found before functions.
unset -f command

# Get a reliable path prefix, handling case where getconf is not
# available.
SYSPATH="$(command -p getconf PATH 2>/dev/null)"
if [[ -z "$SYSPATH" ]]; then
        SYSPATH="/usr/bin:/bin"          # pick your poison
fi
PATH="$SYSPATH:$PATH"
```

这段代码使用了许多非POSIX的扩展，这在14.3节里已说明。

15.2 限制性Shell

限制性Shell（restricted Shell）的设计，是将用户置于严格限制文件写入与移动的环境中，用户多半是使用访客（guest）账号。POSIX并未定义提供限制性Shell的环境，“因为它并未提供历史文件中所暗示的安全性限制”。然而，ksh93与bash两者都提供这一功能，我们将在此介绍它们。

当被引用为rksh时（或使用-r选项）时，ksh93即为限制性Shell。你可以让用户的登录受限制，方法是放置rksh的完整路径名称在用户的/etc/passwd里。ksh93可执行文件必须连接到名为rksh之处，以执行此操作。

限制性ksh93的特定限制不允许用户做下列操作。这些功能有部分是仅ksh93适用，要了解更多信息，可见参考书目中的《Learning the Korn Shell》：

- 变更工作目录：cd是没有作用的。如果你尝试使用它，会收到错误信息ksh: cd: restricted。

- 不允许重定向输出到文件：重定向运算符 >、>|、<>，与 >> 都不被允许。这点不包含 exec 的使用。
- 指定新值给环境变量 ENV、FPATH、PATH 或 SHELL，或试图以 typeset 改变它们的属性。
- 标明任何带有斜杠（/）的命令路径名称。Shell 仅执行在 $PATH 里找到的命令。
- 使用 builtin 命令，增加新的内置命令。

类似于 ksh93 的是：当引用为 rbash 时，bash 即扮演限制性 Shell 的角色，而 bash 可执行文件必须连接到 rbash，以执行此任务。bash 的限制性运算列表和 ksh93 很类似。下面列表里的功能，有部分是 bash 所特有的（参考自 *bash*(1)而来），不过我们不在本书多作介绍。要了解进一步信息，见 *bash*(1)手册页：

- 以 cd 切换目录。
- 设置或解除设置 SHELL、PATH、ENV 或 BASH_ENV 的值。
- 标明含有 / 的命令名称。
- 标明含有 / 的文件名，作为 .（点号）内置命令的一个参数。
- 在内置命令 hash 里使用 -p 选项，指定含有 / 的文件名作为参数。
- 在启动时，自 Shell 环境导出函数定义。
- 在启动时，自 Shell 环境解析 SHELLOPTS 的值。
- 使用 >、>|、<>、>&、&>，与 >> 重定向运算符，重定向输出。
- 使用 exec 内置命令，用另一个命令取代 Shell。
- 以内置命令 enable 搭配 -f 或 -d 选项，增加或删除内置命令。
- 使用 enable 内置命令，启用已停用的 Shell 内置命令。
- 为内置命令 command 标明 -p 选项。
- 使用 set +r 或 set +o restricted 关闭限制性模式。

对这两个 Shell 而言，这些限制都是在用户的 .profile 与环境文件被执行之后才生效。即限制性 Shell 下的用户环境全被设置在 .profile 里。这让系统管理者可以适当地配置环境。

要防止用户覆盖 ~/.profile，只把文件权限设置为用户只读是不够的。根目录不应该被用户写入，或是 ~/.profile 里的命令也不应该 cd 到不同的目录下。

建立这类环境常用的两种方式便是设置“安全”命令的目录，然后让该目录为PATH里的唯一一个，以及设置命令选单。其中，用户没有离开Shell是不能跳离的。无论如何，请确定$PATH下的任何目录中没有其他Shell；否则，用户只要执行该Shell，就能避开先前列的限制。同时，也要确认$PATH下没有任何程序允许用户起始Shell，像是来自ed、ex或vi文本编辑器的“Shell转义（Shell escape）”。

警告： 虽然自原始的Version 7 Bourne Shell起，便拥有限制性Shell的功能，但被使用的很少，因为设置一个可用又正确的限制性环境其实并不容易。

15.3 特洛伊木马

特洛伊木马是看起来无害，有时甚至会误以为它很有用，但却隐藏危险的东西。

想想下面这样的情况：用户John Q.（登录名称为jprog）是一个顶尖的程序设计师，拥有一些个人程序，就放在~jprog/bin里。这个目录出现在~jprog/.profile里PATH变量的第一个。因为他是这样优秀的程序设计师，不久便被提升为系统管理者。

这对他而言是一个全新的领域，而John在不注意的情况下，仍将它的bin目录保留予其他用户可以写入。这时有个居心不良的W.M.先生，建立了这样的Shell脚本，名为grep，放在John的bin目录里：

```
/bin/grep "$@"
case $(whoami) in                              检查有效的用户 ID 名称
root)   nasty stuff here                       危险的操作就放在这!
        rm ~/jprog/bin/grep                    隐匿罪行!
        ;;
esac
```

本质上，当jprog以自已的身份在做事时，这个脚本不会有任何危险。问题出在他使用了su命令之后。su命令可以让一般用户切换到不同的身份。通常用法是：让一般用户成为root（当然，前提是这个用户必须知道密码）。接下来，su会使用它继承的任何PATH设置（注2）。在这里的情况是：PATH包括了~jprog/bin。现在，当jprog以root身份工作，执行grep时，确实执行的是他bin目录下的特洛伊木马版本。这个版本还是会执行真正的grep，所以jprog仍能得到他要的结果。但更重要的是，接下来脚本还会以root身份执行一连串*nasty stuff here*处所指定的命令。即该UNIX会让脚本为所欲为。当一切操作完成，特洛伊木马也删除，不留任何证据。

注2：　使用su - *user*切换用户，就会像用户登录一般，可防止导入已存在的PATH。

可写入的bin目录为特洛伊木马敞开了大门，如同在PATH里具有点号，(想想看，要是root执行cd切换到含有特洛伊脚本的目录中，而且点号是在root的PATH里且位置又先于系统目录时，会发生什么事)。让可写入的Shell脚本放在任何bin目录下更是另一个大门。就好像你晚上会关闭并锁上家门一样，你应该确定关上系统上的任何大门。

15.4 为Shell脚本设置setuid：坏主意

UNIX安全性上的问题有很多是出在它的一个文件属性上，称为*setuid*(设置用户ID)位。这是一个特殊权限位：当一个可执行文件将它打开时，身份会立即转换为与文件拥有者相同的一个有效用户ID。这个有效的用户ID与进程真正的用户ID并不同，UNIX以进程的有效用户ID进行权限检测。

假设你编写了一个游戏程序，可保留私有分数记录文件，显示前15名系统里的玩家。你不希望这个分数文件任何人都能写入，因为这么一来任何人只要动点手脚，就能让自已成为高分的玩家。如果让你的游戏setuid为你的用户ID，则只有你自已拥有的游戏程序可以更新文件，其他人都不行(游戏程序可以通过查看它的真实用户ID来知道谁在执行它，并使用它来决定登录名称)。

setuid工具对游戏与分数文件来说是一个不错的功能，如果设为root时，它就可能变得相当危险。将程序setuid为root，可便于管理者处理需要root权限的文件(例如配置打印机)。为了设置文件的setuid位，只要输入`chmod u+s` *filename*即可。对root拥有的文件设置setuid是很危险的事，所以建议不要在`chown root` *file*后执行`chmod u+s` *file*。

类似的工具程序，在组层级上也有，也就是*setgid*(设置组ID)。`chmod g+s` *filename*即可打开setgid权限。当你执行`ls -l`时，在setuid与setgid的文件上，会出现s权限模式，取代原有的x。例如`-rws--s--x`的文件指的便是拥有者可读取与写入、任何人可执行，且setuid与setgid位都已设置(八进制模式为6711)。

现代系统管理的智慧认为，设置setuid与setgid的Shell脚本是一个可怕的想法。尤其在C Shell下更受影响，因为它的`.cshrc`环境文件有太多可供破坏的地方。而且，它也有很多方式可以将setuid的Shell脚本转化成交互式的Shell，而且是以root的有效用户ID。这就是骇客(cracker)的希望：拥有root执行任何命令的能力。我们从*http://www.faqs.org/faqs/unix-faq/faq/part4/section-7.html*借来一个例子：

> …好，假设有个脚本叫作/etc/setuid_script，一开始是这样：
>
> ```
> #!/bin/sh
> ```
>
> 现在我们来看看假设执行了下面的命令，会发生什么事：

```
$ cd /tmp
$ ln /etc/setuid_script -i
$ PATH=.
$ -i
```

我们知道，最后一个命令将重新安排成：

```
/bin/sh -i
```

因此，此命令会给我们一个交谈模式的Shell，setuid为该脚本的拥有者！幸好这个安全性黑洞可以通过，将第一行指定为：

```
#!/bin/sh -
```

解决掉。将 - 置于选项列表的结尾：则下一个参数 -i 将被视为文件名，正常让命令读取之。

正因为如此，POSIX才容许在 /bin/sh 的选项结尾处使用单一 - 字符。

注意：setuid的Shell脚本与一个setuid Shell之间的差异必须特别留意。后者为Shell可执行版的副本，它是属于 root 并应用setuid位。以上一节的特洛伊木马为例，假定 *nasty stuff here* 部分是这样的：

```
cp /bin/sh ~badguy/bin/myls
chown root ~badguy/bin/myls
chmod u+s  ~badguy/bin/myls
```

还记得，这段代码以 root 身份执行，所以它是可以运作的。当心怀不轨的 badguy 执行了 myls，它是一个机器码的可执行文件，且应用setuid位。待Shell再回到 root 手中时，系统的安全性便荡然无存了。

事实上，setuid与setgid的Shell脚本所带来的危险，在现行UNIX系统上都必须特别留意。包括商用UNIX系统与自由软件（派生自BSD 4.4与GNU/Linux），都停用了Shell脚本上的setuid与setgid位。即便你在文件里应用这些位，操作系统也不会有任何操作（注3）。

我们也发现现在的很多系统加载时可选择是否针对整个文件系统停用setuid/setgid位。这在网络式加载的文件系统上，还有那些可删除式媒体上，例如软驱与光驱，绝对是件好事。

15.5 ksh93与特权模式

Korn Shell特权模式的设计就是为了对付setuid的Shell脚本。这是一个 set -o 选项

注3： Mac OS X与最新的OpenBSD版本是我们发现的两个例外，如果你在这类系统下做事，请特别留意！我们发现Solaris 9只有在文件拥有者非 root 时，才执行setuid的操作。

(set -o privileged 或 set -p)，无论何时当 Shell 执行之脚本已设置 setuid 位时，Shell 便会自动输入它；也就是说，当有效用户 ID 与实际用户 ID 不同时。

在特权模式下，引用一个setuid的Korn Shell脚本时，Shell会执行/etc/suid_profile 文件。此文件应写成限制 setuid Shell 脚本，一如限制性 Shell 那样。至少，它会将 PATH 设为只读（typeset -r PATH 或 readonly PATH），然后设置它为一到多个“安全的”目录。再说一次，这是用以避开引用时的所有陷阱。

因为特权模式是选用的，所以你也可使用 set +o privileged（或 set +p）将它关闭。然而，这对潜在的系统骇客产生不了帮助：Shell 会自动地将它的有效用户 ID 切换为相同的真实用户 ID。也就是说，当你关闭特权模式时，同时也关闭了 setuid。

除特权模式外，ksh 另提供了一个特殊的“代理”程序，会执行 setuid 的 Shell 脚本（或可执行但不可读取的 Shell 脚本）。

为此，脚本的开头不应以 #! /bin/ksh 起始。当程序被引用时，ksh 会试图以正规二进制可执行文件的方式执行程序。当操作系统无法执行脚本（因为它不是二进制的，及因为它没有 #! 标明的解译器名称）时，ksh 会认为它是脚本，及使用脚本的名称与它的参数引用 /etc/suid_exec。除此之外，它还会安排传递一个认证“token”给 /etc/suid_exec，指出脚本的有效用户与组 ID。/etc/suid_exec 会验证执行脚本是否是安全的，再安排以该脚本的适当真实用户与组 ID 引用 ksh。

虽然结合特权模式与 /etc/suid_exec 可以避免很多 setuid 脚本上的攻击，但编写一个可供 setuid 的安全脚本，其实是一门很大的学问，需要很多的知识与经验，应小心对待。

虽然 setuid 的 Shell 脚本在现今系统上不能工作，但有时特权模式也是很好用的。特别是它已广泛应用在第三方所提供的程序 sudo 上，该程序引用自网页上的说法，允许系统管理者给予特定用户（或一群用户），以 root 或另一个用户身份执行部分（或所有）命令的能力，其官方网站为：*http://www.courtesan.com/sudo*。系统管理者如要了解执行管理性工作的环境，只要执行 sudo /bin/ksh -p 即可。

15.6 小结

编写安全的 Shell 脚本也是保全 UNIX 系统安全的一环。本章探讨不过是皮毛，我们建议你深入研究 UNIX 系统安全的相关信息（见“参考书目”）。一开始我们就列出编写安全性 Shell 脚本的提示，这些都是 UNIX 安全性领域的专家所认可的。

接下来介绍的是限制性 Shell，它可以停用许多具潜在危险的操作，其环境构建于用户的

.profile文件里，该文件会在限制性用户登录时执行。实际上，限制性Shell其实很难正确地设置与使用，我们建议你找其他方式设置限制性环境。

特洛伊木马是看似无害但实际上会对系统产生攻击的程序。我们带你看过几种特洛伊木马的建立方式，但其实还有更多。

设置setuid的Shell脚本不是个好主意，几乎所有近期的UNIX系统都已停用它，因为很难关掉它所打开的安全性漏洞。你必须花时间仔细确认你的系统是否已停用它们，如果没有，请定期查找系统里是否还有这类文件。

最后，我们简短地带过Korn Shell的特权模式，它的目的是在解决诸多与Shell脚本相关的安全性议题。

附录A

编写手册页

程序用户需要说明文件，程序设计者同样也需要它，当他们最近很少用到这个软件的时候。可惜的是，很少有计算机书籍提及软件说明文件的制作，所以就算用户想为程序写一份好文件，也不晓得怎么做，甚至不知从何开始。本附录便是为了填补这样的不足。

在UNIX中，简短的程序文件多半为手册页（manual page）的形式，以`nroff/troff`标记（markup）写成（注1），可通过`man`、`nroff -man`或`groff -man`以简单的ASCII文本显示，使用`ditroff -man -Txxx`、`groff -man -Txxx`或`troff -man -Txxx`显示设备xxx的排版方式，或在X windows下以`groff -TX -man`检查。

较冗长的软件说明文件一直以来都是提供使用手册或技术报告，通常是`troff`标记形式，并以PostScript或PDF的形式打印。`troff`标记完全不是以人类看得懂的方式定义，因此，GNU Project选择另一种完全不同的方式：Texinfo文件系统（注2）。Texinfo标记认为比通用的`troff`包更高级，且就像`troff`一样，也允许以简单ASCII文本以及TEX排版系统（注3）检查。最重要的是：它支持超文本链接，让用户在浏览整个在线文档时更方便。

注1：　虽然`nroff`是在`troff`之前开发完成，但从用户的角度来看，这两个系统其实是类似的：`ditroff`与`groff`模拟这两者。

注2：　见Robert J. Chassell与Richard M. Stallman的《Texinfo: The GNU Documentation Format》，由Free Software Foundation于1999年出版，ISBN: 1-882114-67-1。

注3：　见Donald E. Knuth的《The TEXbook》，由Addison-Wesley于1984年出版，ISBN: 0-201-13448-9。

大部分你在UNIX系统里读到的在线文档，可能都是以早期的troff（注4）或Texinfo（注5）标记形成。Texinfo系统里的makeinfo程序可以产生ASCII、HTML、XML与DocBook/XML格式的输出。Texinfo文件可直接由TEX排版，其输出为DVI（device-independent）文件，该文件格式可通过后端DVI驱动程序转成数种设备格式。

当然能用的不只有这些标记格式。Sun Microsystems自Solaris 7开始，即以SGML格式提供（几乎）所有的手册页，而Linux Documentation Project（注6）推动XML（SGML子集）标记，有助于该单位将GNU/Linux文件转换为世界各国语言的目标。

那么，UNIX的程序设计者究竟应使用哪个标记系统呢？经验告诉我们，使用高级标记较佳，即便它较冗长，但绝对值得。SGML（HTML与XML）建立在严谨的语法上，所以在它们编译为可显示的页面之前，文件的逻辑架构仍是很好验证的。有了充分详细的标记，SGML文件即能可靠地转换为其他标记系统，事实上，有些书及杂志出版商正是这么做：作者以任意文件格式交稿，出版商将其转换为SGML，然后再使用troff、TEX或其他排版系统作为后端，产生打印机可读取的页面。

不幸的是SGML软件工具集仍不够充分，且未完整标准化，因此要达到软件文件最大的可移植性，可能还是使用troff或Texinfo标记之一比较好。以手册页来说，如果可以使用man命令，则使用troff格式较佳。

最后，有人仍希望能做出自动化转换两个标记系统的产物，不过这样的目标其实很难做到。你现在能做的，便是用troff标记的限制式子集编写手册页，让它们能自动转换为HTML与Texinfo。要达到此目标，你必须安装两个包：man2html与man2texi（注7）。

pathfind 的手册页

即便完整介绍标记系统文件的书很多，不过你可以从我们这里的介绍，更轻松地学习、了解troff子集。我们在这里会逐步介绍，就像在8.1节里分段介绍pathfind脚本那样，最后会再将这些片段集结成完整的手册页文件，呈现于例A-1。

在开始前，我们先介绍一下nroff/troff标记语言。nroff建置在早期文本格式化系统的经验上，例如DEC的runoff及产生ASCII打印设备的输出结果。当贝尔实验室

注4： 见 *http://www.troff.org/*。

注5： 见 *http://www.gnu.org/software/texinfo/*。

注6： 见 *http://www.tldp.org/*。

注7： 可自 *http://www.math.utah.edu/pub/man2html* 与 *http://www.math.utah.edu/pub/man2texi* 取得。

需要照相凸版排版系统时，troff这个用于产生排版页面的新程序就诞生了。troff为最早期计算机排版的尝试中成功的一个。这两个程序都接受相同格式的输入，所以当我们说troff时，通常也是指nroff。

早期UNIX系统是在极小内存的微型计算机上执行，显然并不适于处理这些格式化程序。troff命令，一如许多UNIX命令，是隐密的与简短的。大部分出现在一行的开头，形式为一个点号，接上一或两个字母或数字。其字体的选择也是有限的：只有roman、粗体、斜体，及后来的等宽字体这几种形式而已。troff的文件里，空格与空白行是有意义的：输入两个空格字符，就会产生（大约）两个输出空格。

然而，简单的命令格式令troff文件的解析更容易，且许多前端处理程序已被开发为提供简单的方程序、图形、图片与表格的规格。它们消耗troff数据流，并产生比原始数据流稍大一些的输出。

虽然完整的troff命令其实内容庞大，但通过-man选项所选定的手册页风格只有一些命令。它不需要前端处理程序，所以手册页里没有方程或图片，表格也很少。

手册页文件的版面配置相当简单，六七行标准的顶层标题段落，穿插着一些文本的格式化段落，及缩进的、定标签的区块。就像你每次使用man命令所看到的那样。

手册页的检查方式，长久以来累积了相当多种类，在显示的文体上也有很大的不同，当标记是视觉的而非逻辑的时候较容易被预期。我们在此选择的字体，只是建议性，而非强制一定要使用。

现在是开始编写pathfind手册页的时候了，这是一个相当简单的程序，因此它的标记不致太难处理。

我们从注释性语句开始，因为每个计算机语言都应该要有。troff的注释从反斜线引用（backslash-quote）开始，直至结尾，但不包含end-of-line。当它们紧接在初始的点号之后，它们的行终结符也会从输出中消失：

```
.\" ============================================================
```

由于troff输入不能被缩进处理，所以它看起来非常密集。我们发现，在标头段落之前的等号注释行会让它们比较好辨识，且我们时常使用相当短的输入行。

每个手册页文件都以*Text Header*命令（.TH）开始，其至多可带有4个参数：大写命令名称、手册段落编号（数字的1为用户命令），以及可选用的再版日期与版本编号。这些参数用以构建执行中的页面标头与格式化后输出文件的页尾：

```
.TH PATHFIND 1 "" "1.00"
```

Section Heading（部分标题）命令（.SH）则只带有一个参数，如含有空格，请引用它。且请遵循手册页惯例，以大写字母表示：

```
.\" ============================================================
.SH NAME
```

NAME段落的主体，提供的是apropos（等同于man -k）命令所需的要件，它应该只有一行，结尾不带有任何标点符号。形式为command - description：

```
pathfind \(em find files in a directory path
```

标记\(em是手册页里可见的少数几个troff命令之一：它表示一个em -（破折号），也就是大约是字母m宽度的水平行。前置一个空格并且接着em -（破折号）。较旧的手册页用的是\-（负号），或只是-，但em -（破折号）为英语印刷样式的惯用法。

第二个段落为在命令行引用程序时，提供的简短概要说明。一开始仍为标头：

```
.\" ============================================================
.SH SYNOPSIS
```

接下来有时会是漫长的标记，最常出现的就是字体信息：

```
.B pathfind
[
.B \-\^\-all
]
[
.B \-\^\-?
]
[
.B \-\^\-help
]
[
.B \-\^\-version
]
```

选项-（连字号）以\-标记，以取得负号的排版方式，看起来会比稍短的原始连字号好。我们使用\^命令，防止在troff的输出中将连字号一起执行。nroff的输出下，空格字符会消失。程序名称与选项被设置为粗体字。字体转换的命令，像.B，可使用到6个参数（如它们包含空格，请引用它），然后每一个都紧邻着排版。当出现多个参数时，意即所有字间需要的空格都应该明确地提供。在此，方括号为默认的roman字体；在手册页中，它们界定可选用的值。虽然我们应该将关闭与开启的方括号置于同一行，但我们不这么做，因为让每个选项可以在三个连续行上完成可便于编辑。字体配对的命令虽可立即接上，让它们变成单唯一行，但它们很少被用在选项列表里。

除了断行，troff会以塞满段落的模式排版，因此所有东西看起来只有一行。以经验来

说，我们发现nroff的ASCII输出会在--version选项之后断行，但因为我们是在段落模式下，所以下一行会从最左边缘接上。这部分有点讨厌，所以我们只有在套用nroff时置入条件语句处理，但在troff里就不需要这么使用。这里是以临时缩进（temporary indentation）命令（.ti）加上参数+9n处理，即缩进9个空格符，大约是命令名称的宽度加上等宽字体的结尾空格符：

```
.if n .ti +9n
```

命令行很短，放在单一排版行上绰绰有余，所以对troff无须再作这类处理。这里是它大致的样式，但我们隐藏了注释，待程序加入了更多的选项，我们再添上：

```
.\" .if t .ti +\w'\fBpathfind\fP\ 'u
```

缩进总数的计算很复杂，因为它与字体成比例，且我们无法得知命令名称的宽度。\w'...'u的命令是在计算单引号里元素的宽度。因为文本被设置为粗体字，所以我们使用内部的字体包装：\fB...\fP，即转换为粗体字后，再转换回原先的字体。类似的字体转换命令还有roman（\fR）、斜体字（\fI），与等宽（\fC）字体。C表示的是Courier，这是广为流传的等宽字体。

接下来的命令行处理为：

```
envvar [ files-or-patterns ]
```

第三段落描述程序的选项。这部分置于所有进一步说明之前，是因为大部分在手册页里，它是最常读取的段落：

```
.\" ============================================================
.SH OPTIONS
```

部分选项会接上简短的备注说明，所以接下来处理这个：

```
.B pathfind
options can be prefixed with either one or two hyphens, and
can be abbreviated to any unique prefix.  Thus,
.BR \-v ,
.BR \-ver ,
and
.B \-\^\-version
are equivalent.
```

这个段落展现了一个新特色：成对字体命令（.BR），这里设置其参数是粗体与roman字体的文本之间没有任何空格。类似的命令还有：.IR与.RI斜体-roman配对，.IB与.BI粗体-斜体配对，当然还有已经介绍过的.RB。不过等宽字体并没有类似的用法，因为它是后来才加入的（原始的贝尔实验室排版程序并没有这样的字体），你必须改用\fC...\fP。

现在该是切开段落的时候了：

```
.PP
```

在nroff的输出中，一空行与一段落切分是一样的意思，但troff则使用少数的垂直空格作为段落切分。在段落之间使用.PP会是比较好的形式，一般来说，手册页输入文件绝不应含有任何空行。

接下来的段落如下：

```
To avoid confusion with options, if a filename begins with a
hyphen, it must be disguised by a leading absolute or
relative directory path, e.g.,
.I /tmp/-foo
or
.IR ./-foo .
```

现在，我们可以开始进行选项描述了。它们的标记应该算是用在手册页里最复杂的部分，不过要上手也很快。本质上，我们要的是有标签的（labeled）缩进段落，辅以段落第一行最左边的标签设置。近期许多标记系统均以项目列表构建此部分：起始－选项－列表、起始－选项、结束－选项、起始－选项、结束－选项等等，然后以结束－选项－列表作终结。不过，手册页标记不全然这么做，它只是起始于项目，但终结于下一个段落切分（.PP）或部分标头（.SH）。

起始项目的命令（.TP）可选择性地设置宽度参数，设置描述性段落从左边缘开始的缩进宽度。如参数省略，则使用默认的缩进。如标签长度大于缩进，则新的行立即自标签之后开始。段落缩进仍会影响接下来的.TP命令，所以只有选项列表里的第一个需要它。如同使用SYNOPSIS部分里封装的命令行的缩进，我们使用动态的缩进，根据最长选项名称的长度而定。由于我们有好几个选项要说明，所以这里以具有一连串破折号的注释行将之区分：

```
.\" ---------------------------------------------------------
.TP \w'\fB\-\^\-version\fP'u+3n
```

接在.TP命令之后的行会提供项目标签：

```
.B \-all
```

标签之后接的是选项描述：

```
Search all directories for each specified file, instead of
reporting just the first instance of each found in the
search path.
```

如果这个描述需要切分段落，则使用缩进段落（Indented Paragraph）命令（.IP）取代

原来的段落切分命令（.PP），这么做不会终结此列表。不过这份手册页很短，我们用不到 .IP。

接下来的选项描述就不再需要用到新的标记了，下面为完整的选项部分：

```
.\" -----------------------------------------------------------
.TP
.B \-?
Same as
.BR \-help .
.\" -----------------------------------------------------------
.TP
.B \-help
Display a brief help message on
.IR stdout ,
giving a usage description, and then terminate immediately
with a success return code.
.\" -----------------------------------------------------------
.TP
.B \-version
Display the program version number and release date on
.IR stdout ,
and then terminate immediately with a success return code.
```

手册页第4段为程序描述。这部分的长度由你决定：Shell能执行到数十页。然而，我们期待它是简短的，因为手册页时常会被查阅。pathfind相当简单，只要三段就能描述完成。前两段的标记是我们已经知道的：

```
.\" ===========================================================
.SH DESCRIPTION
.B pathfind
searches a colon-separated directory search path defined by
the value of the environment variable, \fIenvvar\fP, for
specified files or file patterns, reporting their full path on
.IR stdout ,
or complaining \fIfilename: not found\fP on
.I stderr
if a file cannot be found anywhere in the search path.
.PP
.BR pathfind 's
exit status is 0 on success, and otherwise is the number of
files that could not be found, possibly capped at the
exit code limit of 125.
.PP
```

最后一小部分是必须了解的手册页标记，显示在最后一段。在此，我们要以计算机输入与输出的等宽的缩进方式显示，而非一般填满段落的方式。字体的改变，是类似我们先前所提到的 \fC...\fP。当它出现于行起始时，我们会前置 troff 的 no-op 命令 \&，因为如果接下来的内文一开始就是点号时，就必须使用no-op。我们要计算机范例是缩

进的，所以将缩进范围边界以 *Begin Right Shift*（.RS）与 *End Right Shift*（.RE）命令界定。并且，我们还需要停止填满整个段落，所以在内文前后加上 *no fill*（.nf）与 *fill*（.fi）命令：

```
For example,
.RS
.nf
\&\fCpathfind PATH ls\fP
.fi
.RE
reports
.RS
.nf
\&\fC/bin/ls\fP
.fi
.RE
on most UNIX systems, and
.RS
.nf
\&\fCpathfind --all PATH gcc g++\fP
.fi
.RE
reports
.RS
.nf
\&\fC/usr/local/bin/gcc
/usr/bin/gcc
/usr/local/gnat/bin/gcc
/usr/local/bin/g++
/usr/bin/g++\fP
.fi
.RE
on some systems.
.PP
Wildcard patterns also work:
.RS
.nf
\&\fCpathfind --all PATH '??tex'\fP
.fi
.RE
reports
.RS
.nf
\&\fC/usr/local/bin/detex
/usr/local/bin/dotex
/usr/local/bin/latex
/usr/bin/latex\fP
.fi
.RE
on some systems.
```

最后部分提供其他相关命令的交叉引用信息，这些信息对读者可能相当有用，所以请彻

底执行。它的格式很简单：只是一个以字母顺序排列的单一段落，且其命令名称为粗体并辅以使用手册部分编号，各命令以逗点隔开，最后以点号结束：

```
.\" ======================================================
.SH "SEE ALSO"
.BR find (1),
.BR locate (1),
.BR slocate (1),
.BR type (1),
.BR whence (1),
.BR where (1),
.BR whereis (1).
.\" ======================================================
```

我们几乎已经介绍完所有常见的手册页标记了。唯一的重要遗漏便是*Subsection Heading*命令（`.SS`），不过它很少见，只出现在较冗长的手册页文件里，其运行与`.SH`命令类似，只不过它在排版输出中使用较小的字体。来自`nroff`的ASCII输出，在视觉上并无差异。另有两个行内命令，有时你可能会需要用到`.\|.\|.`表示省略符号（即...），与`\(bu`表示项目标记（即·），时常作为以下标签段落列表中的标签，像这样：

```
.TP \w'\(bu'u+2n
\(bu
```

至此已检查过手册页的分析。完整的`troff`输入，我们收集在例A-1，而排版后的输出（来自`groff -man`，默认产生PostScript）则显示于图A-1。有了我们的指南，你应该可以开始着手编写程序的手册页了。

例 A-1：pathfind 的 troff 手册页标记

```
.\" ======================================================
.TH PATHFIND 1 "" "1.00"
.\" ======================================================
.SH NAME
pathfind \(em find files in a directory path
.\" ======================================================
.SH SYNOPSIS
.B pathfind
[
.B \-\^\-all
]
[
.B \-\^\-?
]
[
.B \-\^\-help
]
[
.B \-\^\-version
]
```

PATHFIND(1) PATHFIND(1)

NAME

pathfind — find files in a directory path

SYNOPSIS

pathfind [**--all**] [**--?**] [**--help**] [**--version**] envvar [file(s)]

OPTIONS

pathfind options can be prefixed with either one or two hyphens, and can be abbreviated to any unique prefix. Thus, **-v**, **-ver**, and **--version** are equivalent.

To avoid confusion with options, if a filename begins with a hyphen, it must be disguised by a leading absolute or relative directory path, e.g., */tmp/-foo* or *./-foo*.

-all Search all directories for each specified file, instead of reporting just the first instance of each found in the search path.

-? Same as **-help**.

-help Display a brief help message on *stdout*, giving a usage description, and then terminate immediately with a success return code.

-version Display the program version number and release date on *stdout*, and then terminate immediately with a success return code.

DESCRIPTION

pathfind searches a colon-separated directory search path defined by the value of the environment variable, *envvar*, for specified files, reporting their full path on *stdout*, or complaining *filename: not found* on *stderr* if a file cannot be found anywhere in the search path.

pathfind's exit status is 0 on success, and otherwise is the number of files that could not be found, possibly capped at the exit code limit of 125.

For example,

```
pathfind PATH ls
```

reports

```
/bin/ls
```

on most Unix systems, and

```
pathfind --all PATH gcc g++
```

reports

```
/usr/local/bin/gcc
/usr/bin/gcc
/usr/local/gnat/bin/gcc
/usr/local/bin/g++
/usr/bin/g++
```

on some systems.

SEE ALSO

find(1), **locate**(1), **slocate**(1), **type**(1), **whence**(1), **where**(1), **whereis**(1).

1.00 1

图 A-1：pathfind 排版后的手册页

```
.if n .ti +9n
.\" .if t .ti +\w'\fBpathfind\fP\ 'u
envvar [ files-or-patterns ]
.\" ========================================================
.SH OPTIONS
.B pathfind
options can be prefixed with either one or two hyphens, and
can be abbreviated to any unique prefix.  Thus,
.BR \-v ,
.BR \-ver ,
and
.B \-\^\-version
are equivalent.
.PP
To avoid confusion with options, if a filename begins with a
hyphen, it must be disguised by a leading absolute or
relative directory path, e.g.,
.I /tmp/-foo
or
.IR ./-foo .
.\" --------------------------------------------------------
```

```
.TP \w'\fB\-\^\-version\fP'u+3n
.B \-all
Search all directories for each specified file, instead of
reporting just the first instance of each found in the
search path.
.\" ----------------------------------------------------------
.TP
.B \-?
Same as
.BR \-help .
.\" ----------------------------------------------------------
.TP
.B \-help
Display a brief help message on
.IR stdout ,
giving a usage description, and then terminate immediately
with a success return code.
.\" ----------------------------------------------------------
.TP
.B \-version
Display the program version number and release date on
.IR stdout ,
and then terminate immediately with a success return code.
.\" ==========================================================
.SH DESCRIPTION
.B pathfind
searches a colon-separated directory search path defined by
the value of the environment variable, \fIenvvar\fP, for
specified files or file patterns, reporting their full path on
.IR stdout ,
or complaining \fIfilename: not found\fP on
.I stderr
if a file cannot be found anywhere in the search path.
.PP
.BR pathfind 's
exit status is 0 on success, and otherwise is the number of
files that could not be found, possibly capped at the
exit code limit of 125.
.PP
For example,
.RS
.nf
\&\fCpathfind PATH ls\fP
.fi
.RE
reports
.RS
.nf
\&\fC/bin/ls\fP
.fi
.RE
on most UNIX systems, and
.RS
.nf
```

```
\&\fCpathfind --all PATH gcc g++\fP
.fi
.RE
reports
.RS
.nf
\&\fC/usr/local/bin/gcc
/usr/bin/gcc
/usr/local/gnat/bin/gcc
/usr/local/bin/g++
/usr/bin/g++\fP
.fi
.RE
on some systems.
.PP
Wildcard patterns also work:
.RS
.nf
\&\fCpathfind --all PATH '??tex'\fP
.fi
.RE
reports
.RS
.nf
\&\fC/usr/local/bin/detex
/usr/local/bin/dotex
/usr/local/bin/latex
/usr/bin/latex\fP
.fi
.RE
on some systems.
.\" =======================================================
.SH "SEE ALSO"
.BR find (1),
.BR locate (1),
.BR slocate (1),
.BR type (1),
.BR whence (1),
.BR where (1),
.BR whereis (1).
.\" =======================================================
```

手册页语法检查

检查手册页的格式化是否正确，通常是视觉地直接看，你只要使用下面其中一个命令，打印输出即可：

```
groff -man -Tps pathfind.man | lp
troff -man -Tpost pathfind.man | /usr/lib/lp/postscript/dpost | lp
```

或者使用这样的命令，在屏幕上以 ASCII 或是排版输出：

```
nroff -man pathfind.man | col | more
groff -man -Tascii pathfind.man | more
groff -man -TX100 pathfind.man &
```

col 命令会处理由 nroff 针对平行的与垂直的操作所产生的特殊转义符，不过 groff 的输出就不需要用到 col 了。

有些 UNIX 系统里会提供简单的语法检查程序：checknr，命令为：

```
checknr pathfind.man
```

它在我们的系统上不会产生警告信息。checknr的功能是找出不符合的字体，但对手册页格式的了解不多。

很多 UNIX 系统里都有 deroff，它是一套简单的过滤程序，用以去除 troff 标记。你可以像这样执行拼字检查，：

```
deroff pathfind.man | spell
```

为了避免来自拼字检查程序对于troff标记错误的抱怨。找出文件里难以察觉的问题还有两个好用的工具：叠词查找（doubled-word finder，注 8）与界定符平衡检查（delimiter-balance checker，注 9）。

手册页格式转换

转换为 HTML、Texinfo、Info、XML 与 DVI 文件很简单：

```
man2html pathfind.man
man2texi --batch pathfind.man
makeinfo pathfind.texi
makeinfo --xml pathfind.texi
tex pathfind.texi
```

碍于长度，我们不在这里展现 .html、.texi、.info 与 .xml 文件的输出。如果你好奇可以自已做做看，再一窥其内容，了解它们的标记格式。

手册页的安装

一直以来，都是使用 man 命令，在环境变量 MANPATH 定义的查找路径下之各个子目录中 —— 通常像这样 /usr/man:/usr/local/man，寻找手册页。

注 8： *http://www.math.utah.edu/pub/dw/*。

注 9： *http://www.math.utah.edu/pub/chkdelim/*。

有些近期的 man 版本只是假设，在程序查找路径 PATH 中的每个目录，可以放置 /../man 字符串在尾端，以找出相对应的手册页目录，而不需用到 MANPATH。

在各个手册页目录下，通常是寻找前置 man 与 cat，及结尾是部分编号的成对子目录。在各子目录内，文件名也以部分编号作为结尾。因此 /usr/man/man1/ls.1 为 ls 命令文件的 troff 文件，而 /usr/man/cat1/ls.1 则保存 nroff 的格式化输出。man 使用的是后者，当它存在时，可避免重新执行不必要的格式化。

当有部分厂商采用全然不同的手册页树状结构组织时，它们的man实例仍能认得过去存在过的实现方式。因此，大部分 GNU 软件的安装将可执行文件置于 $prefix/bin 中，并将手册页置于 $prefix/man/man1，其中 prefix 默认为 /usr/local，且在各类系统下都能运行得当。

系统管理者通常会在固定一段期间安排执行catman或makewhatis，以更新来自手册页 NAME 部分中含有单行描述的文件。该文件是给 apropos、man -k 与 whatis 命令所使用的，目的在于提供手册页的简单索引。如果这么做还找不到你想要的东西，就只能求助于 grep，使用全文查找了。

附录B

文件与文件系统

如果想要有效率地利用计算机，那么对文件与文件系统就要有基本的了解。本附录要呈现的是 UNIX 文件系统重点功能的概要：什么是文件？文件如何命名、包括了哪些东西？如何将它们层级式地聚集在文件系统里？它们有哪些特性？

什么是文件

简单的说，一个文件就是存在于计算机系统里的一堆数据，而且可以用单一实体的方式从计算机程序中引用。文件还提供让进程执行可以继续的数据存储机制，一般常用于重新启动计算机（注 1）。

早期计算机上，文件属于计算机系统外部的东西：多半是放在磁带、纸带（paper tape），或打孔卡（punched cards）上。谁拿着卡片就能管理里面的文件，想用它的人只要愿意把一大叠打孔卡从地上抱起来就可以。

过了一段时间，磁碟就开始广泛使用。它们的物理大小一直在缩减，从整只手臂的大小缩到只有你拇指的宽度，不过它们的容量却是一直在长大，从 20 世纪 50 年代年中期的 5MB，到 2004 年的 400 000MB。成本与访问时间已经至少下降了 3 倍。而今，能使用的磁碟种类已经相当多了。

注 1：部分系统将特殊的快速文件系统置放在中央的随机访问内存（random-access memory，RAM）里，让进程之间得以共享临时文件。以一般 RAM 技术来说，这类文件系统需搭配不断电系统，因为它们常会在系统重启后重建一个新的。然而，有些嵌入式计算机系统（embedded computer system）使用非挥发性（nonvolatile）的 RAM，可提供长期的文件系统。

光学存储设备，例如 CD-ROM 与 DVD，已经是廉价又高容量的选择了：20 世纪 90 年代，CD-ROM 大举取代软盘（flexible magnetic disk；floppy）与商用软件发布所使用的磁带。

另外可以用的还有非挥发性的固态（solid-state）存储设备；它们最后应该会取代某些具有移动机制部分的设备，后者会因为逐渐耗损而失效。不过在编写本书的时候，成本上的考虑远大于它的替代性，它们不但容量比较小，且仅能重写几次而已。

文件如何命名

早期计算机操作系统无法命名文件：文件的处理是由它们的所有者提出，再由人工计算机操作人员一次处理一个。不久马上就有人发现，如果文件能自动地处理就更好了：文件需要人类可用来归类与管理的名称，而计算机也可使用该名称识别它们。

当我们可以指定名称给文件后，马上就会发现必须处理名称冲突的问题，因为很可能出现相同名称指定予两个或更多个不同文件的情况。现代文件系统解决这个问题的方法是将独一无二的文件名逻辑式地组合在一起，称为目录（directory）或文件夹（folder）。这部分在本附录稍后“UNIX 层级式文件系统”里将有所介绍。

在文件命名时，使用的是从主机操作系统里字符集取得的字符。早期的计算上，字符集各有相当大的差异，但因为必须在相异的系统间交换数据便凸显了标准化的需求。

1963 年，*American Standards Association*（注 2）以冗长的 *American Standard Code for Information Interchange* 名称，提出 7 位字符集，之后即以其初始字母 ASCII（发音为 *ask-ee*）广为流传。7 个位允许呈现 $2^7 = 128$ 个不同的字符，已经足以处理拉丁字母的大写与小写、数字与一些特殊符号及标点符号字符，包括空格以及剩下的 33 个控制字符。后者未指定可打印的图形表现方式。它们之中，有些是用作将行加以标示与分页，但大部分是用于特定用途。几乎所有计算机系统都已支持 ASCII。想了解 ASCII 字符集，请使用命令 `man ascii`。

不过，这么多的世界语言，使用 ASCII 表示是不够的：它能贮藏的字符太少了。因为大部分的系统现在都使用 8 位字节，作为最小的定址存储单位，它允许 $2^8 = 256$ 个不同字符。系统设计师也立即将该 256 元素集合的上半部分拿来使用，将 ASCII 留在下半部分。可惜的是他们未遵循国际标准，所以出现了几百种不同的各种字符指定方式，有时它们会被称为内码页（code page）。即使单一 128 个额外字符集的空间，对完整的欧洲语系

注 2：之后重新命名为 *American National Standards Institute* (*ANSI*)。

仍嫌不足，因此*International Organization for Standardization* (*ISO*)便开发了一系列代码页（或称内码页）：ISO 8859-1（注3）、ISO 8859-2、ISO 8859-3等。

20世纪90年代，共同开发的单一万国字符集Unicode（注4）开始运作。这最终需要每个字符大约有21个位，但许多操作系统下的现行实例只使用到16个位。UNIX系统使用一个可变动的位宽度编码：*UTF-8*（注5），允许已存在的ASCII文件成为有效的Unicode文件。

会讨论字符集上是因为：除了独特的IBM大型计算机使用*EBCDIC*（注6）字符集外，所有现行系统都将ASCII字符集纳入128以下的位置。因此将文件名限制在ASCII子集，我们就可以让这个名称通用于所有地方了。现在的Internet与World Wide Web便证明了文件可以在不同的系统间进行交换。

原始的UNIX文件系统设计者，决定这256个元素集合都可用于文件名，但有两个例外：一个是控制字符NUL（此字符的所有位都为零），这是许多程序语言里，用来表示字符串结尾的字符；另一个则是斜杠（/）字符，这是用来保留重要用途的字符，稍后会介绍。

此选择是相当宽容，不过我们强烈建议你再加强进一步的限制，理由如下：

- 因为文件名是人们也要使用，所以它的名称必须是可视字符：看不到的控制字符不适合。
- 文件名不单单是人类要用，计算机也要用：人们可从上下文认出作为文件名的字符字符串，但计算机程序需要更精确的规则。
- 在文件名里使用Shell的meta字符（也就是大部分的标点符号）必须特殊处理，因此最好都避免。
- 初始的连字号会让文件名看起来像UNIX命令的选项。

注3：　可到*http://www.iso.ch/iso/en/CatalogueListPage.CatalogueList*查找ISO Standards目录。

注4：　《The Unicode Standard, Version 4.0》，由Addison-Wesley于2003年出版，ISBN: 0-321-18578-1。

注5：　见*RFC 2279: UTF-8*，《a transformation format of ISO 10646》，见*ftp://ftp.internic.net/rfc/ rfc2279.txt*。

注6：　EBCDIC = Extended Binary-Coded Decimal Interchange Code，发音为*eb-see-dick*或*eb-kih-dick*。这是一套在1964年首次出现在IBM System/360系统上的8位字符，包括了旧式6位的IBM BCD集合作为子集。System/360及其后来的产物都是应用在计算机上最久的架构，许多全球企业都使用它们。IBM也支持使用ASCII字符集的GNU/Linux实例，见 *http://www.ibm.com/linux/*。

部分非UNIX文件系统允许在文件名里使用大、小写字符，但在比较文件时却会忽略大小写的不同。UNIX原始的文件系统则不是这样，对它们来说：readme、Readme，与README是三个不同的文件名（注7）。

UNIX文件名惯用的方式是全为小写，因为它好读、好输入。某些常见的重要文件名，例如AUTHORS、BUGS、ChangeLog、COPYRIGHT、INTALL、LICENSE、Makefile、NEWS、README，与TODO则惯用大写或混用大小写。因为大写字母在ASCII字符集里位于小写字母之前，因此这些文件在进行目录列表时，会一开始就出现，而更容易被看到。然而，以现行UNIX系统而言，排序顺序视locale而定，所以将环境变量LC_ALL设为C，即可得到传统ASCII的排序方式。

为了在其他操作系统上也能使用，最好是将文件名限制在拉丁字母的字符、数字、连字号、下划线及单一个点号。

文件名可以多长？这根据文件系统而定，而有些软件本身的缓冲区有固定大小，限制了所能处理的最大文件名。早期UNIX系统有14个字符的限制。但从20世纪80年代中期，UNIX系统的设计便普遍允许使用到255个字符。POSIX定义了NAME_MAX常数为该长度，不包含终结的NUL字符，且要求最小值为14。X/Open Portability Guide要求最小值为255。你可以使用getconf（注8）命令找出你系统的限制。下面是你在大部分UNIX系统里会看到的报告：

```
$ getconf NAME_MAX .                在当前的文件系统下，文件名可以多长 ？
255
```

文件位置的完整规格有另一个且更大的限制，我们会在本附录“文件系统架构”部分提及。

警告： 我们在这里，对使用空格字符于文件名中的做法提出警告。有些窗口式的桌面环境操作系统，其文件名是从滚动菜单中被选定，或输入到对话方块中，这让它们的用户会以为在文件名里使用空格字符是没问题的。其实不是!文件名不单只是在这个小小的对话框里被用到，唯一明智的做法应是在有限的字符集里选择字符，作为你的文件名。特别是UNIX Shell，它的命令是可以使用空格字符加以分隔的。

因为文件名里可能出现空白或其他特殊字符，在Shell脚本里，你应该要记得将任何可能含有文件名的Shell变量的计算总是以引号括起。

注7： Mac OS X里支持的旧式HFS式文件系统会视大小写为相同，所以将软件移植到该系统上，可能会出现意料之外的情况。Mac OS X也支持一般视大小写为不同的UNIX文件系统。

注8： 几乎所有UNIX系统里都有，除了Mac OS X与FreeBSD（5.0前的版本）外。getconf的源代码可以在glibc的发布包里找到：*ftp://ftp.gnu.org/gnu/glibc*。

UNIX 的文件里有什么

UNIX 另一个了不起的成就，就是以简单的观点来看文件：UNIX 的文件，不过是零或多个不知名的数据字节集结而成的流。

大部分的其他操作系统将文件看成两种：二进制与纯文本的数据、计数长度（counted-length）与固定长度与可变长度的记录、索引与随机与顺序的访问，等等。这马上就成了一种梦魇：简单的复制文件操作，可能就因为文件类型的不同而必须以不同的方式完成，且必须所有软件都能处理数种文件，复杂度会更高。

UNIX 的文件复制其实没什么：

```
try-to-get-a-byte
 while (have-a-byte)
 {
    put-a-byte
    try-to-get-a-byte
 }
```

这种循环的排序可被实例在许多程序语言中，它最棒的地方是在：程序无须知道数据从哪里来，它可以从文件、磁带设备、管道、网络连接、内核数据结构，或是从任何未来设计者所设计的数据来源而来。

你会说，那我需要一个特殊文件，文件尾端有一个具指针的目录指向稍早的数据，且该数据本身是加密的。在 UNIX 中的答案是：没问题！你只要让应用程序了解你这个完美的文件格式，完全不会带给文件系统或操作系统该复杂度。它们不必了解这些细节。

然而，UNIX 仍对允许的文件稍作区分。人为建立的文件，通常含有数行文本，以分行字符作为结尾，且不会出现无法打印的 ASCII 控制字符。这样的文件可以被编辑、显示于屏幕上、被打印、以电子邮件传送，还能通过网络传递到其他计算机系统上，其数据也保证是被维护的很完整。用于处理文本文件的程序，包括我们在本书讨论的诸多软件工具，其设计上是使用大的但大小固定的缓冲区来保存文本行。如果给它们过长的行，或具有无法打印的字符的输入文件，则它们可能会出现无法预知的行为。处理文本文件时，建议你将行的长度限制在读取时较舒适的大小，例如 50 到 70 个字符。

文本文件以 ASCII linefeed（LF）字符，在 ASCII 表里为十进制值 10，表示行的界线。此字符指的是换行字符。许多程序语言在字符字符串里，以 `\n` 表示此字符。这种表示方式，比其他系统的一组 carriage-return/linefeed 字符的表示方式简单多了。在 C 与 C++ 程序语言里的广泛用法以及后来开发的一些语言，都以单一换行字符作为文本文件里每个行的终结；这是由于它们有很多都是源自于 UNIX。

在共享文件系统的混用操作系统环境下，常会需要为使用不同行结尾符的文本文件作转换。*dosmacux* 包（注 9）提供了一组方便的工具来做这件事，同时保留了文件的时间戳。

UNIX 里所有的其他文件可被认为是二进制文件：每一个包含在其中的字节，都有 256 种可能的值。因此，文本文件可以算是二进制文件的子集。

不同于某些操作系统的是，没有字符会被抢夺以表示 end-of-file：UNIX 文件系统单纯地在文件中保留字节数的计数。

尝试读取超越文件字节计数时，则返回一个 end-of-file 的暗示，所以它不可能看到任何磁盘块之前的内容。

有些操作系统禁用空文件，但 UNIX 不这么做。有时，它表示的只是一个文件的存在，重点不在它的内容。例如时间戳、文件锁定，以及 `THIS-PROGRAM-IS-OBSOLETE` 这样的警告，都是有用的空文件范例。

UNIX 将文件视为字节流的想法，鼓励操作系统的设计者，实现出看起来像文件但非传统式文件想法的数据。许多 UNIX 版本，实现一个进程信息虚拟文件系统（pseudofilesystem）：只要输入 `man proc` 就可以知道你系统提供的有哪些。我们在 13.7 节里已做过详细的讨论。在 `/proc` 树状结构下的文件，并未真实存在于大型存储设备下，它只是提供察看进程表格及执行中进程的内存空间，或了解操作系统内部信息（例如处理器、网络、内存与磁盘系统）的详细数据的方式。

以我们写本书时所使用的系统为例，我们可以找到与存储设备相关的细节，类似这样（命令参数上，斜杠表示的意义将在下节讨论）：

```
$ cat /proc/scsi/scsi                          显示磁盘设备信息
Attached devices:
Host: scsi0 Channel: 00 Id: 00 Lun: 00
  Vendor: IBM      Model: DMVS18V          Rev: 0077
  Type:   Direct-Access                    ANSI SCSI revision: 03
Host: scsi1 Channel: 00 Id: 01 Lun: 00
  Vendor: TOSHIBA  Model: CD-ROM XM-6401TA Rev: 1009
  Type:   CD-ROM                           ANSI SCSI revision: 02
```

UNIX 层级式文件系统

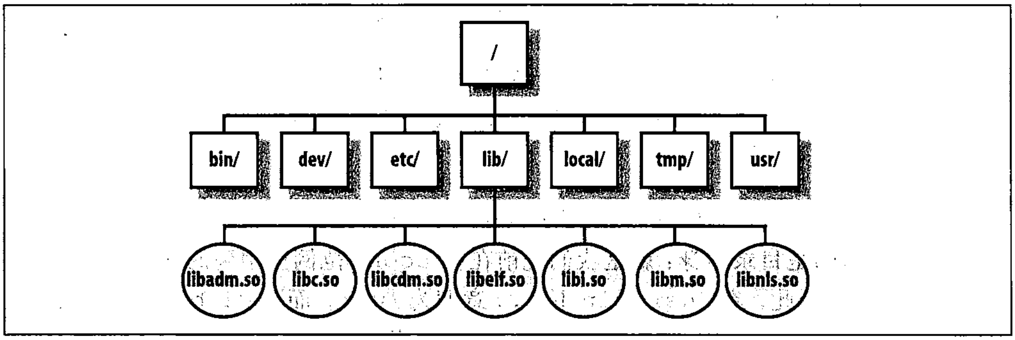
大量的文件就可能有文件名冲突的风险，如果要所有名称都独一无二，管理上也相当困难。UNIX 处理的方式，便是将文件组织在目录（directory）下：每个目录形成它自已

注 9： *http://www.math.utah.edu/pub/dosmacux/*.

的名称空间，独立于所有其他的目录。目录也可提供默认属性给文件，这个主题我们将在稍后的“文件所有权与权限”部分做简短介绍。

文件系统架构

目录可以嵌套配置为任意深度，使得UNIX文件系统形成树状结构。UNIX在此不使用文件夹（folder）是因为纸本文件的文件夹无法嵌套配置。文件系统目录结构的本源为根目录（root directory），它有一个特殊而简单的名称：/（ASCII的斜杠）。/myfile指的就是根目录下，叫作myfile的文件名称。斜杠还有另一个目的：用来界定名称，以记录目录的嵌套架构。图B-1呈现的是文件系统顶层架构的一小部分。



图B-1：文件系统目录结构

UNIX目录下可包括任意数目的文件。不过，大部分现行UNIX文件系统的设计与文件系统程序界面，都假定目录是被连续地查找。因此在大型目录下寻找文件的时间，便与目录里的文件数成比例。当文件超过百个，最好以子目录重新组织。

在嵌套式目录的完整列表下，要到达一文件，是以路径名称（pathname）或称为路径的方式引用。它有时会包含文件名本身，有时则不会，视当时的情况而定。文件名的完整路径，包含名称本身，能有多长？一直以来，UNIX的文件都未提供答案，但POSIX定义PATH_MAX常数来限制其长度，包含终结的NUL字符，要求最大值为256，但X/Open Portability Guide则要求到1024。你可以使用getconf命令查询你系统里的限制，以我们的系统为例：

```
$ getconf PATH_MAX .                    在当前文件系统下，路径名称的最大长度为何?
1023
```

其他我们测试过的UNIX系统，也有报告1024或4095的。

C程序语言的ISO标准称此值为FILENAME_MAX，且它必须定义在标准标头文件stdio.h

里。我们检查过许多UNIX版本，还发现255、1024，与4095的值。Hewlett-Packard HP-UX的10.20与11.23的值只有14，但它们的getconf报告则为1023与1024。

因为UNIX系统支持多个文件系统，文件名长度也为文件系统的特性之一，与操作系统无关，所以编译期常数所定义的这些限制是没有意义的。高级语言的程序员多半被建议使用pathconf()或fpathconf()函数库调用，以取得这些限制值：它们需要传递一个路径名称，或是一个打开的文件描述代码，使得特定的文件系统可以被识别。也就是为什么我们在先前的例子里，传递当前的目录（点号）给getconf。

UNIX目录本身就是文件，只不过它拥有特殊属性且限制性访问。所有UNIX系统都包括顶层目录bin，保存（时常是二进制）可执行程序，包括很多在本书中使用过的那些。这个目录的完整路径名称为/bin，它很少包含子目录。

另一个普遍性顶层目录为usr，不过它一定含有其他目录，/usr/bin就是其中一个。它和/bin是不同的，本附录稍后的“文件系统实现概况”会说明，如何让两个bin目录看起来一样（注10）。

所有的UNIX目录，就算是空的，也至少包括两个特殊目录：.（点号）与..（点号点号）。第一个指的是目录本身：就是我们先前在getconf范例里用到的那个；第二个指的则是父目录。因此，在/usr/bin下，..意即为/usr，而../lib/libc.a意思就是/usr/lib/libc.a——这是C语言执行期程序库存放的惯例位置。

根目录的父目录就是自已，所以/、/..、/../..、/../../..都是一样的。

路径结尾如果以斜杠结束，则它是一个目录。如果最后字符非斜杠，那么最后一个组成部分是目录还是其他类型的文件，则只能咨询文件系统而得知。

POSIX要求路径里连续的斜杠被视为单一斜杠。这要求在我们参考到最早期的UNIX文件里并未明白指定，但自20世纪70年代中期起，Version 6源代码一开始，即完成此斜杠减少（注11）。因此：/tmp/x、/tmp//x，与//tmp//x都指同一个文件。

注10： DEC/Compaq/Hewlett-Packard OSF/1 (Tru64)、IBM AIX、SGI IRIX，与Sun Solaris都做得到。Apple Mac OS X、BSD系统、GNU/Linux，与Hewlett-Packard HP-UX则不能做到。

注11： 见John Lion的书：《Lions' Commentary on UNIX 6th Edition, with Source Code》，1996年由Peer-to-Peer Communications出版，ISBN 1-57398-013-7。此修正出现在kernel行7535 (sheet 75)，注释说明于p.19-2的“Multiple slashes are acceptable”。如果程序码以if取代while，则此减少不会发生。

在这本书里，有很多的注脚提供World Wide Web的来源位置（URL），其语法是以UNIX的路径名称所形成。URL 前置通信协议的名称与主机名称，例如：proto://host，指的即为 UNIX 风格的路径名称，置于主机的网页目录树下。网页服务器需要这些信息，找到它们在文件系统里的适当位置。URL 自 20 世纪 90 年代晚期开始广泛使用，使得 UNIX 路径名称为人们所熟悉。

层级式文件系统

如果斜杠为根目录，则每个文件系统里只会有一个，那么 UNIX 要如何支持多个文件系统，但又可以避免根目录名称冲突的情况呢？答案很简单：UNIX允许将某个文件系统，逻辑性地置于另一个文件系统内一个已存在的任意目录之上。该操作称为加载（mounting），相关命令为 mount 与 umount：分别为加载与卸载文件系统。

当另一个操作系统加载在一个目录之上时，该目录先前的内容都无法看见也无法访问，只有在卸载以后它们才会再出现。

文件系统加载会让人觉得单一文件系统树会无限长大的幻觉，只需通过简单地加入更多或更大的存储设备即可。正规的文件名称惯例 /a/b/c/d/... 即指出对用户与软件而言，无须关心其设备为何，这点不同于其他操作系统：后者会将设备名称放置在路径名称之中。

完成加载命令需要充分的信息，因此系统管理员将这些细节存储在一个特殊文件里，通常是 /etc/fstab 或 /etc/vfstab，视 UNIX 的版本而定。该文件一如大部分的 UNIX 组态文件：都为一般文本文件，其格式可参考手册页 *fstab*(4 或 5) 或 *vfstab*(4)。

当共享的磁盘是唯一可用的文件系统媒体时，加载与卸载便需要特殊权限，通常只有系统管理员可以做这件事。不过，对一些个人拥有的媒体，例如软盘、光盘或DVD，桌上计算机的用户需要能够自己做这件事。许多 UNIX 系统进行了功能的扩充，所以有某些设备也允许非特权用户进行加载与卸载。这里是自 GNU/Linux 系统下使用的例子：

```
$ grep owner /etc/fstab | sort                    哪些设备允许用户加载 ？
/dev/cdrom    /mnt/cdrom      iso9660 noauto,owner,kudzu,ro 0 0
/dev/fd0      /mnt/floppy     auto    noauto,owner,kudzu 0 0
/dev/sdb4     /mnt/zip100.0   auto    noauto,owner,kudzu 0 0
```

这里设置让用户可以使用 CD-ROM、软盘，与 Iomega Zip，使用方式如下：

```
mount /mnt/cdrom                      使光驱呈可用状态
cd /mnt/cdrom                         改变到它的顶层目录
ls                                    列出其文件
...
cd                                    改变根目录
umount /mnt/cdrom                     释放光驱
```

mount 命令不使用参数与特殊权限时：只会简单地报告所有当前加载的文件系统。下列为独立的网页服务器范例：

```
$ mount | sort                                显示已加载的文件系统列表，并排序它
/dev/sda2 on /boot type ext3 (rw)
/dev/sda3 on /export type ext3 (rw)
/dev/sda5 on / type ext3 (rw)
/dev/sda6 on /ww type ext3 (rw)
/dev/sda8 on /tmp type ext3 (rw)
/dev/sda9 on /var type ext3 (rw)
none on /dev/pts type devpts (rw,gid=5,mode=620)
none on /dev/shm type tmpfs (rw)
none on /nue/proc type proc (rw)
none on /proc/sys/fs/binfmt_misc type binfmt_misc (rw)
none on /proc type proc (rw)
```

这里显示，根文件系统加载在磁盘设备 /dev/sda5 下。其他文件系统则分别加载为 /boot、/export 等等。

系统管理员可使用来下命令卸载 /ww 树：

```
# umount /ww                                  在此，# 为根提示符号
```

如果 /ww 目录下有任何文件正在使用，则此命令的执行结果会失败。你可以使用 lsof (list-open-files) 命令（注 12），以追踪正被防止卸载的进程。

文件系统实现概况

文件系统实现的细节很有趣，但也太复杂，且超出这本书的范畴。我们建议你参考更好的书，例如《The Design and Implementation of the 4.4BSD Operating System》（注 13）与《UNIX Internals: The New Frontiers》（注 14），进一步了解。

从较高层的观点来看文件系统实现其实是相当有帮助的，因为这么做可以从用户的角度去看 UNIX 的文件系统。文件系统建立时，一个管理员指定的固定大小表格（注 15）也随之建立，以保存文件系统中与文件相关的信息。每个文件都会与此表格的一个实现产生相关，每个实现都为一个文件系统数据结构，被称为 *inode*（*index node* 的缩写，发

注 12： *ftp://vic.cc.purdue.edu/pub/tools/UNIX/lsof/*。其他 UNIX 版本下的替代命令可使用 fstat 与 fuser。

注 13： 作者 Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, 与 John S. Quarterman，于 1996 年由 Addison-Wesley 出版，ISBN 0-201-54979-4。

注 14： 作者 Uresh Vahalia，于 1996 年由 Prentic-Hall 出版，ISBN 0-13-101908-2。

注 15： 部分高级文件系统设计允许根据需要加大表格。

音为*eye node*)。*indoe*的内容视特定的文件系统设计而定，因此单一系统下可能含有数种不同的形式。程序员都被隔绝在 stat()与 fstat()系统调用的差异之外（见 *stat*(2)手册页)。参考 man inode 可以了解你系统上实际结构的相关信息。

由于inode结构与存储设备的其他低层细节都与系统息息相关，因此通常不太可能将某厂商的UNIX文件系统加载在另一个厂商的UNIX文件系统下。不过我们可以通过软件 *Network File System（NFS）*解决这个问题，它可以跨网络共享各个不同厂商所提供的UNIX文件系统。

由于inode表格为固定大小，因此有可能出现文件系统已满，但存储设备仍有大量可用空间的情况：还有空间可用来置放文件的数据，而没有空间放它的*metadata*（数据的数据)。

如图B-2所示，inode条目包括了系统辨认文件时所需的所有数据，只有一件事除外：它的文件名。这似乎很令人惊讶，事实上，是有许多使用类似文件系统设计的操作系统将文件名包括在类似于inode的条目中。

Number	Type	Mode	Links	Byte count	User ID	Group ID	Disk address	Attributes
0	...	...	...	...	...	...	...	...
1	...	...	...	...	...	...	...	...
2	...	...	...	...	...	...	...	...
3	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...

图 B-2：Inode 表格内容

在UNIX下，文件名伴随其inode编号存储在目录里，如图B-3所示。早期20世纪70年代小型计算机里的UNIX系统，仅在目录下，为每个文件配置16个字节：2个字节给inode编号（文件编号限制为$2^{16} = 65\ 536$)，剩下的14个字节则给文件名使用，只比一些其他系统的8+3限制好一点。

现代UNIX文件系统允许较长文件名，不过传统上还是有最大长度的限制，请参考本附录先前“文件系统架构”提供的 getconf 范例。

对目录的所有者，及早期一些需打开与读取目录以寻找文件的UNIX软件而言，目录只能读取，不能写入。更复杂的目录设计在20世纪80年代问世，opendir()、readdir()与closedir()程序库调用的建立，让程序员看不到它们的架构，这些调用也成为现在POSIX的一部分（见 *opendir*(3)手册页)。为加强程序库的访问，部分现行的UNIX实现，禁止在目录文件上进行读取运算。

i-Node number	Filename
2155329	.
737046	..
1294503	ch04.xml
2241988	README
3974649	Makefile
720277	ch04.ps
2945369	CVS
523023	CH-AA-SHELL-EXTENSIONS.txt
351882	ch04.xml.~1~
...etc...	...etc...

图 B-3：目录表格内容

注意： 为什么 UNIX 要将文件名与剩下的文件 metadata 加以分隔呢？理由至少有两个：

- 通常用户列出目录内容的目的，只是为了提醒自已：文件是不是就在这个目录下。如果文件名存在inode里，当你在目录下寻找各个文件名时，可能得访问磁盘一到多次。将名称存储在目录文件里，再多的名称都只需自单一磁盘块里取出即可。
- 如果文件名与inode各自独立，则同一个物理文件，就能拥有数个文件名，只需通过不同的目录条目引用到相同inode即可。这些引用甚至不需要在同一个目录里！这是文件别名的概念，在 UNIX 下称之为连接（link），是一个相当方便且广为使用的功能。在 6 种不同的 UNIX 版本下，我们就发现 /usr 下有 10%~30% 的文件为连接。

UNIX 文件系统设计里，还有一个很有用的结果就是重新命名文件或目录，或是在同一个 UNIX 实现文件系统内移动它，速度都很快：只有名称需要改变或移动，而不会动到内容。在文件系统之间移动文件，则需要对文件所有块进行读取与写入的操作。

如果文件有数个名称，那么哪一个才能删除文件？应该在删除后，所有名称立即消失，还是只有某一个被删除？这部分是由文件系统的设计师决定支持别名还是连接；UNIX 选择后者。UNIX的inode条目包括了连接到文件内容的计数。文件删除将引发连接计数的减少，但只有计数为零时，文件块最终才会重新指派给可用空间的列表。

因为目录条目包括的只是inode数字，所以它只可以引用同一个物理文件系统内的文件。我们已知道UNIX文件系统通常会包含数个加载点，我们怎么才能在这个文件系统里做一个连到另一个文件系统里的连接？解决方式是使用另一种连接：软连接（soft link），或称为符号连接（symbolic link），有时直接称为 *symlink*，这是为了与第一种硬连接

(hard link) 加以区别。符号连接表示的是“该目录条目指向另一个目录条目”(注 16)，而非 inode 条目。指向的条目（pointed-to entry）为一般 UNIX 路径名称，因此可以指向文件系统内任何位置，就算是跨加载点也可以。

符号连接也表示可能在文件系统里产生无穷循环的可能性，为防止这样的情况发生，如需制作一连串的连接，请不要超过 8 个（传统上）。以下为两个元素的循环：

```
$ ls -l                                       显示连接循环
total 0
lrwxrwxrwx   1 jones    devel             3 2002-09-26 08:44 one -> two
lrwxrwxrwx   1 jones    devel             3 2002-09-26 08:44 two -> one
$ file one                                    one 是什么文件?
one: broken symbolic link to two
$ file two                                    two 是什么文件?
two: broken symbolic link to one
$ cat one                                     试着显示 one 文件
cat: one: Too many levels of symbolic links
```

基于技术上的理由（其中一个可能会造成循环），目录通常不能有硬连接，但可以有符号连接。此规则的例外就是 . 与 .. 目录，它们在目录被建立时即自动地产生。

设备作为 UNIX 文件

UNIX 另一个先进的做法，便是将文件的概念延展到系统上的设备。所有的 UNIX 系统都拥有名为 /dev 的顶层目录，在该目录下，则是一些难懂的文件名，例如 /dev/audio、/dev/sda1 与 /dev/tty03。这些设备文件由特殊软件模块控制，也就是设备驱动程序 (device driver)，它会知道如何与特定的外部设备进行沟通。虽然设备名称会因系统的不同而有很大的差异，但它们的功能都是提供一个类似文件的“打开－处理－关闭”访问模式。

注意： 将设备整合到层级式文件系统里，是 UNIX 最棒的点子（The integration of devices into the hierarchical file system was the best idea in UNIX.）。

—— Rob Pike et al.,《The Use of Name Spaces in Plan 9》, 1992.

/dev 树里的实体以特殊工具 mknod 所建立，该工具通常隐藏在 MAKEDEV 这个 Shell 脚本里，且需要系统管理员权限才能执行：见 *mknod*(1) 与 *MAKEDEV*(8) 的手册页。

大部分 UNIX 的用户很少需要引用 /dev 树下的成员，不过有两个例外：/dev/null 与 /dev/tty，这些我们在 2.5.5.2 节里已做过介绍。

注 16： 在 inode 中的文件类型会记录文件是符号连接，且在大部分的文件系统设计中，它指到的文件名称会被存储在符号连接的数据块中。

20 世纪 90 年代，一些 UNIX 版本引进随机伪设备（random pseudodevice），/dev/urandom，作为永远非空的随机字节流。许多密码与安全性软件，需要这样的数据来源。我们在第 10 章已经展示过，使用 /dev/urandom 构建一个难猜的临时文件名。

UNIX 文件到底可以多大

UNIX的文件大小通常受限于两个硬性条件：在inode条目里所分配到的位数，用来保存文件大小（以字节计），以及文件系统本身的大小。除此之外，有些UNIX内核还提供管理员设置文件大小的限制。大部分UNIX文件系统所使用的数据结构会在一个文件中记录数据块列表，加诸的限制是大约 168 万个块，其中一个块的大小基本上是 1 024 到 65 536个字节，可在文件系统建立期被设置且固定住。最后，而文件系统备份设备的容量，也可能会加上更进一步与站台相关的限制。

没有名称的文件

UNIX 操作系统另一个特点，就是打开供输入或输出的文件名称，不会被保留在内核的数据结构中。因此，在命令行上针对标准输入、标准输出或是标准错误输出而被重定向的文件名，都不为被引用的进程所知。想想看：我们的文件系统里已经有几百万个文件了，这三个没有名称也没有不好！不过为了弥补这个缺陷，近期有些 UNIX 系统提供了这样的名称 /dev/stdin、/dev/stdout 与 /dev/stderr，或是比较难记的 /dev/fd/0、/dev/fd/1 与 /dev/fd/2。GNU/Linux 与 SunSolaris 还支持 /proc/PID/fd/0。下面我们可以来看看你的系统是否支持它们；你要不就是执行成功，如下所示：

```
$ echo Hello, world > /dev/stdout
Hello, world
```

要不就是失败，如下所示：

```
$ echo Hello, world > /dev/stdout
/dev/stdout: Permission denied.
```

很多UNIX程序发现它们在重定向文件时需要名称，所以一般惯例是使用连字号作为文件名，使用连字号不表示文件名为连字号；而是指标准输入或标准输出，视上下文而定。我们在这里强调这是习惯用法，因为并非所有 UNIX 软件都如此应用。如果你就是不喜欢这样的文件，你也可以前置目录名称伪装它，例如 ./--data。部分程序遵循惯例，详见2.5.1 节使用双连字号选项 --，表示命令行自此之后是一个文件，而非一个选项，不过这种方式依然并非统一用法。

大部分现行UNIX文件系统都使用32位整数，以保存文件大小，且由于文件定位系统调用，可以在文件中前后移动，因此该整数必须带有正负号。最大可能的文件大小为 2^{31} - 1个字节，约为2GB（注17）。到了20世纪90年代初期，许多磁盘设计都小于此数字，但在2000年时磁盘却能容纳100GB甚至更大的空间，甚至还能将数个磁盘结合成单一逻辑性磁盘，所以现在才能有更大型的文件系统。

UNIX厂商已渐渐升级至可处理64位的文件系统上，即能支持至8亿GB。但你想想，如果写了一个这么大的文件，以当前合理的执行速度10MB/s来看，这个文件要执行27800年！这个移植绝对会产生相当大的影响，因为所有现行使用"随机访问文件定位系统调用"的软件都必须被更新。为避免这种大规模的升级，很多厂商仍允许在较新系统上使用旧式的32位大小，只要不超过2GB限制即可正常运作。

UNIX文件系统建立时，基于效能上的理由会保留一小部分空间，通常是10%，给由`root`执行的进程使用。文件系统本身所需的inode表格空间，通常是放在特殊的初级块，只供磁盘控制器硬件可以访问。因此，磁盘有效空间通常只有磁盘厂商估计的80%。

有些系统里会提供降低这些保留空间的命令：在大型磁盘上，我们会建议你使用它。在BSD及商用UNIX系统上，你可以参考*tunefs*(8)手册页，GNU/Linux的系统则可参考*tune2fs*(8)。

内置Shell命令`ulimit`可控制系统资源的限制。`-a`选项将显示所有资源的值。在我们的系统上，结果如下：

```
$ ulimit -a                                    显示当前用户的进程限制
...
file size (blocks)          unlimited
...
```

你的系统可能会由于本地管理的政策不同而有不一样的结果。

在某些UNIX站台，磁盘限制是启动的（详见*quota*(1)的手册页），它可以进一步限制单一用户所能使用的文件系统空间总量。

UNIX文件属性

本附录稍早，在"文件系统实现概况"的部分曾提及UNIX文件系统的实现，并说明inode的条目记录中包括了*metadata*：除了名称之外，有关文件的相关信息。现在我们要讨论的就是这些属性，因为它们与文件系统的用户息息相关。

注17：GB = gigabyte，约十亿字节。在计算机里，使用G如果非度量衡单位，即表示 2^{30} = 1,073,741,824。

文件所有权与权限

或许，与单一用户的个人计算机文件系统比起来，UNIX 文件的最大不同之处就在于所有权（ownership）与权限（permissions）了。

所有权

在很多的个人计算机上，任何的进程或用户都能读取或写入所有文件，因此计算机病毒现在对读者来说非常熟悉。但是因为UNIX用户能访问的文件系统是受到限制的，所以要替换或破坏重要的文件系统元件很难：病毒很少对 UNIX 系统造成问题。

UNIX文件有两种所有权（或称所有权）：用户（user）与组（group），它们各有自己的权限。一般来说，文件的所有者应具完整的访问权，而该所有者的工作组的成员拥有的权限会有些许限制，除此之外的其他人，权限就再更少一些。前述最后的类别，在UNIX的文件里称之为其他人（other）。文件所有权可使用 ls 命令的冗长模式来显示。

新的文件通常会继承其创造者的所有者与组成员，如果要给予适当的权限，则只有系统管理员通过 chown 与 chgrp 命令改变它们的属性。

在inode条目记录中，用户与组都以数字识别而非名称。因为人们通常偏好以名称识别，因此系统管理员提供对应的表格，一直以来我们都称为密码文件：/etc/passwd与组文件 /etc/group。在大型站点，这些文件多半会替换为某种网络分布式的数据库形式。这些文件或是数据库，任何登录的用户都可读取，不过现今偏向使用程序库调用 setpwent()、getpwent()与endpwent()访问密码数据库、使用 setgrent()、getgrent()与endgrent()访问组数据库：参考 *getpwent*(3)与 *getgrent*(3)的手册页。如果你的站点使用数据库取代 /etc 下的文件，你可以试试Shell命令：ypcat passwd 检查密码数据库，或ypmatch joines passwd寻找用户jones的条目记录。如果你的站台使用NIS+而非NIS，则yp命令应改为niscat passwd.org_dir与nismatch name=jones passwd.org_dir。

重点部分是通过用户与组标识符的数字值来控制访问。如果一文件系统通过用户smith以 user ID 100 被加载或导入，则一个文件系统的 user ID 100 指定给用户 jones，那么 jones 便能完整访问 smith 的文件。就算目标系统下还有另一个 smith 用户也一样。这类的考虑在大型组织的UNIX文件系统下就相当重要了，因它面向全局性可访问的UNIX文件系统：用户与组的识别必须涵盖整个组织范围，是必须的考虑。问题不是只有这里讲的这么简单：用户与组的标识符也有诸多限制。旧式UNIX系统仅能为每一个配置 16 位，也就是总计为 2^{16} = 65 536 个值。较新的 UNIX 系统则允许 32 位的标识

符，遗憾的是，它们有许多都被加诸严格的限制，大大地限制了标识符的数目，这些数字对大型企业而言仍嫌不足。

权限

UNIX文件系统权限有三种类型：读取（read）、写入（write）与执行（execute）。它们每一个在inode数据结构里都只需要单一位，即可指出权限的存在与否。它们会分别针对用户、组，与其他人设置权限。文件权限可通过`ls`命令的冗长模式显示，通过`chmod`命令变更。因为权限每个设置都只需要三个位，因此它可以单一八进制（注18）数字表示，`chmod`命令也接受3个或4个八进制数字的参数或符号形式。

chmod

语法

`chmod [ options ]` *mode file(s)*

主要选项

`-f`

强制变更，如果可能的话（如果失败，不要显示信息）。

`-R`

将变更递归地应用到整个目录。

用途

变更文件或目录的权限。

行为模式

必需的参数*mode*，可以是绝对性的3个或4个八进制数字之一个权限掩码，或是一或多个字母的符号表示：`a`（全部，同于`ugo`）、`g`（组）、`o`（其他人）、或`u`（用户），再接上`=`（设置）、`+`（加入），或`-`（除去），最后则是一或多个`r`（读取）、`w`（写入），或`x`（执行）。多符号的设置需以逗点分隔，因此，755的模式，等同于`u=rwx,go=rx`、`a=rx,u+w`与`a=rwx,go-w`。

警告

递归的形式是相当危险，请谨慎使用！它可能会因误用`chmod -R`应用，而需要从备份媒体中恢复整个文件树。

注18：BSD系统例外：它们提供*sappnd*与*uappnd*标志，可使用`chflags`设置之。

注意： 有些操作系统支持额外的权限。其中有个很有用、但UNIX没有的权限，叫作附加权限（注19）：它在日志文件上特别好用，可用来确保数据只能被加入，但现存的数据不能被更改。当然，如果此文件能被删除，就能再替换为变更数据后的文件，所以附加权限提供的安全性只是错觉。

默认权限的设置会应用至每一个新建立的文件：它们由umask命令控制，以给定的参数设置默认值，如未提供参数，则直接显示默认值。umask的值为三个八进制数字，表示要被拿走的权限：通常值为077，指的是给用户完整的权限（读取、写入、执行），而组与其他人都不具任何权限。其结果为新建立之文件，限制在只有拥有它们的用户可以访问。

现在让我们来看看这些文件权限：

```
$ umask                                          显示当前的权限掩码
2
$ touch foo                                      建立一个空文件
$ ls -l foo                                      列出与文件相关的信息
-rw-rw-r--    1 jones    devel                   0 2002-09-21 16:16 foo
$ rm foo                                         删除文件
$ ls -l foo                                      再次列出与文件相关的信息
ls: foo: No such file or directory
```

一开始，权限掩码为2（确切说法为002），即删除其他人的写入权限。touch只是更新文件最后写入的时间戳，如有需要时建立文件。ls -l命令为冗长式文件列出的惯用语法。它报告了-的文件类型（一般文件）与权限字符串rw-rw-r--（指的是用户与组具读取与写入权限，其他人则具读取权限）等信息。

我们将掩码改为023之后重建文件，以删除组的写入权限与其他人的写入与执行的权限。会看到这样的权限字符串：rw-r--r--，也就是我们所预期的：删除组与其他人的写入权限：

```
$ umask 023                                      重设权限掩码
$ touch foo                                      建立空文件
$ ls -l foo                                      列出文件相关信息
-rw-r--r--    1 jones    devel                   0 2002-09-21 16:16 foo
```

权限运作

什么是执行权限？文件通常不具此权限，除非它们是可以执行的程序或脚本。通常这类程序的连接器都会自动地加上执行权限，不过脚本不会，我们得自行使用chmod变更。

注19： 默认权限。

在复制一个拥有执行权限的文件，例如 /bin/pwd，该权限会被保留，除非 umask 的值使得它们被删除：

```
$ umask                                            显示当前的权限掩码
023
$ rm -f foo                                        删除任何存在的文件
$ cp /bin/pwd foo                                  复制一份系统命令
$ ls -l /bin/pwd foo                               列出文件相关信息
-rwxr-xr-x    1 root     root       10428 2001-07-23 10:23 /bin/pwd
-rwxr-xr--    1 jones    devel      10428 2002-09-21 16:37 foo
```

最后结果 rwxr-xr-- 反映部分权限的消失：组的写入访问消失、其他人的写入与执行也不存在。

最后，我们使用符号形式的参数执行 chmod，为所有人加入执行权限：

```
$ chmod a+x foo                                    为所有人加入执行权限
$ ls -l foo                                        列出冗长式文件信息
-rwxr-xr-x    1 jones    devel      10428 2002-09-21 16:37 foo
```

最后的权限字符串为 rwxr-xr-x：用户、组与其他人均可执行。这里要注意的是，权限掩码不会对 chmod 操作造成影响：掩码只在文件建立的时候有影响。至于复制的文件，行为模式和原始的 pwd 命令一样：

```
$ /bin/pwd                                         尝试系统版本
/tmp
$ pwd                                              以及 Shell 内置版本
/tmp
$ ./foo                                            还有我们对系统版本所复制的
/tmp
$ file foo /bin/pwd                                查看这些文件的信息
foo:       ELF 32位 LSB executable, Intel 80386, version 1,
           dynamically linked (uses shared libs), stripped
/bin/pwd:  ELF 32位 LSB executable, Intel 80386, version 1,
           dynamically linked (uses shared libs), stripped
```

请注意我们在引用 foo 时加上目录前置字符：基于安全性理由，绝不要在 PATH 列表里包括当前目录。如果你一定要这么做，也请你将它放在最后一个！

警告： 如果你试过上述这些，在试图执行 /tmp 下的命令时，可能会得到 permission-denied 的回应。在提供这样功能的系统上，如 GNU/Linux，系统管理员有时会以没有执行权限的模式加载这个目录（tmp）；请检查 /etc/fstab 下是否有 noexec 选项。使用这个选项的另一个理由是为了避免特洛伊木马脚本（参考第 15 章）在像 /tmp 这样公开可写入的目录下被执行。你仍然可以将它们放入 Shell 中而执行，但是你需要知道为什么要这么做。

下面是你删除可执行权限又试图执行程序时，会发生的事：

```
$ chmod a-x foo                              删除所有人的可执行权限
$ ls -l foo                                  列出冗长式文件信息
-rw-r--r--    1 jones    devel         10428 2002-09-21 16:37 foo
$ ./foo                                      试图执行程序
bash: ./foo: Permission denied
```

这里指的不是文件是否有像可执行程序一样的执行能力(ability),而是是否拥有执行权限(possession of execute permission),决定了它能否像命令一样被执行。这是UNIX里一个很重要的安全功能。

当你提供执行权限给不应该具有此权限的文件时:

```
$ umask 002                                  删除默认的都可写入权限
$ rm -f foo                                  删除任何已存在的文件
$ echo 'Hello, world' > foo                  建立一个单行文件
$ chmod a+x foo                              使之可执行
$ ls -l foo                                  显示我们做的变更
-rwxrwxr-x    1 jones    devel            13 2002-09-21 16:51 foo
$ ./foo                                      试图执行程序
./foo: line 1: Hello,: command not found
$ echo $?                                    显示退出状态码
127
```

Shell会要求内核执行 `./foo` 及得到失败报告,其使用设置为 `ENOEXEC` 的程序库错误指示器。Shell接下来会试着自已执行它。在命令行上 `Hello, world` 被解释为命令 `Hello`、参数 `world`。因为在查找路径下找不到这样的命令,所以Shell报告一连串的错误信息,并回传127退出状态码,详见6.2节。

检查权限时,依序为用户、组,最后才是其他人。它们是由所属的进程决定该设置哪些权限位。因此很可能文件属于你,但你却不能读,而你的组成员及系统里的其他人却可以。像这样:

```
$ echo 'This is a secret' > top-secret     建立单行文件
$ chmod 044 top-secret                     对组与其他人,删除所有权限只保留读取权限
$ ls -l                                    显示我们的变更
----r--r--    1 jones    devel             17 2002-10-11 14:59 top-secret
$ cat top-secret                           试着显示文件
cat: top-secret: Permission denied
$ chmod u+r top-secret                     允许所有者读取文件
$ ls -l                                    显示我们的变更
-r--r--r--    1 jones    devel             17 2002-10-11 14:59 top-secret
$ cat top-secret                           这时,便能显示了!
This is a secret
```

所有UNIX文件系统都另提供额外的权限位:*set-user-ID*、*set-group-ID* 与 *sticky*(粘连)位。为兼容旧系统并避免增加已存在的行长度,`ls` 不使用三个额外的权限字符来显示这些权限,而是将 `x` 改为其他字母。详见 *chmod*(1)、*chmod*(2)与 *ls*(1)手册页。基于安全性理由,Shell脚本绝不应该设置 `set-user-ID` 或 `set-group-ID` 权限位:我们发

现太多这类脚本里可怕的安全性漏洞。这些权限位与Shell脚本的安全性议题，在第15章已说明过。

有时我们会在商用软件上应用仅允许执行的权限（--x--x--x），以禁止复制、除虫，与追踪操作，但程序仍可以执行。

目录权限

现在为止，我们讨论的都是一般文件的权限。在目录上，这些权限的解读会稍有不同。目录的读取，即能列出它的内容，例如使用ls。写入，则表示你能在目录下建立或删除文件，即便你对目录下的文件不具写入权限：该特权保留给操作系统，以维持文件系统的一致性。执行访问，即你可以访问文件以及该目录下的子目录（当然受它们自己权限的管制），特别是你还可以跟随该目录下的路径名称。

由于目录的执行与读取较难区分，我们在这里举例解释：

```
$ umask                                         显示当前的权限掩码
22

$ mkdir test                                    建立子目录
$ ls -Fld test                                  显示目录权限
drwxr-xr-x  2 jones devel 512 Jul 31 13:34 test/

$ touch test/the-file                           建立空目录
$ ls -l test                                    目录内容冗长式列出
-rw-r--r--  1 jones devel 0 Jul 31 13:34 test/the-file
```

至此，都为一般行为模式。现在，我们删除读取权限，但留下执行权限：

```
$ chmod a-r test                                删除所有人读取目录的权限
$ ls -lFd test                                  显示目录权限
d-wx--x--x  2 jones devel 512 Jan 31 16:39 test/

$ ls -l test                                    试图列出目录内容
ls: test: Permission denied

$ ls -l test/the-file                           列出文件本身
-rw-r--r--  1 jones devel 0 Jul 31 13:34 test/the-file
```

第二个ls失败是因为缺乏读取权限，但因为有执行权限，所以第三个ls成功。这里呈现的是：删除目录的读取权限并不能防止目录下的文件被访问，用户只要知道文件名就可以这么做。

当我们删除执行访问，却未恢复读取权限时：

```
$ chmod a-x test                                删除所有人执行目录的权限
$ ls -lFd test                                  列出目录
d-w-------  3 jones devel 512 Jul 31 13:34 test/
```

```
$ ls -l test                                  试图列出目录内容
ls: test: Permission denied

$ ls -l test/the-file                         试图列出文件
ls: test/the-file: Permission denied

$ cd test                                     试图改变目录
test: Permission denied.
```

目录树不再允许所有用户浏览，root 除外。

最后我们恢复读取，但不要恢复执行访问，再重复刚刚做的事：

```
$ chmod a+r test                              加入所有人读取目录的权限
$ ls -lFd test                                显示目录权限
drw-r--r--  2 jones devel 512 Jul 31 13:34 test/

$ ls -l test                                  试图列出目录内容
ls: test/the-file: Permission denied
total 0

$ ls -l test/the-file                         试图列出文件
ls: test/the-file: Permission denied

$ cd test                                     试图改变目录
test: Permission denied.
```

缺乏对目录的执行权限，是无法浏览它的内容，或是不能使它成为当前的工作目录。

目录设置黏着位时，里头所含的所有文件就只有它们的所有者或目录所有者才能删除。此功能最常用在公用的可写入目录，例如 /tmp、/var/tmp（过去的 /usr/tmp）这些，还有邮件进来的目录，以防止用户删除不属于他们的文件。

某些系统上，目录设置 set-group-ID 位时，新建立的文件的组 ID 即为此目录的组 ID 而非所有者所属组。可惜的是，这样的权限位并非在所有系统下都如此处理。另有一些系统，其行为模式需视加载的文件系统为何而定，所以你应该在你系统里，再确认一次 mount 命令的手册页。当有好几个用户协同开发项目时，set-group-ID 位的设置就相当好用了。我们可以为此项目建立一个特殊的组，并建立成员，然后再将项目的目录设置给该组。

部分系统结合 set-group-ID 位的设置与 group-execute 位，这种用法太过复杂，已超出本书范围，在此不进行介绍。

目录读取与执行权限

为什么读取目录与通过该目录至其子目录有不同的含义？答案很简单：它是为了在看不到父目录的情况下，仍能看到子目录下的文件。最常见的使用就是在用户的网页结构下。根目录通常为`rwx--x--x`这样的权限，防止组或其他人列出目录内容或检查文件。但网页目录的起始，假设是`$HOME/public_html`，包含其子目录，我们可以给予它们`rwxr-xr-x`这样的权限，且在那之下的文件，则至少拥有`rw-r--r--`的权限。

另一个例子是，假设为了安全性的理由，系统管理员想要对先前未防护的文件子目录进行读取保护（read-protect）。他需要做的就只是删除该子目录顶层的根目录（单一目录）之读取与执行权限`chmod a-rx` *`dirname`*即可；这使所有这之下的文件都立即无法新的打开（但已打开的则不受影响），即便他们拥有个别文件的使用权限。

注意： 有些UNIX系统支持访问控制列表（access control lists，ACL）。它可以提供较细的访问控制，可针对个别的用户与组指定非默认的权限。可惜的是，ACL工具的设置与显示在各系统间都不尽相同，使其难以在异构环境中使用，在本书中做进一步讨论也不适当。如果你想了解更多，可以试着使用`man -k acl`或`man -k 'access control list'`，在你的系统下查找相关的命令。

文件时间戳

UNIX文件的inode条目记录包括三个重要时间戳：访问时间、inode变更时间与修改时间。这些时间一般是自*epoch*（注20）算起的秒数计之，epoch的UNIX系统时间为00:00:00 UTC, January 1, 1970，不过有些UNIX实现提供更好的计时单位。以UTC（注21）[国际标准时间（Coordinated Universal Time），早期为格林威治时间（Greenwich Mean Time，GMT）] 计算的时间，表示该时间戳不受本地时区设置影响。

访问时间是通过数个系统调用而被更新，包括那些读取与写入文件的操作。

注20： **epoch**，ep'ok，名词。用以编号之后年度的一个固定时间点。

注21： 经委员会一致通过：UTC为不受语言影响的字母缩略字，法文的展开为Temps Universel Coordonné。见*http://www.npl.co.uk/time/time_scales.html*、*http://aa.usno.navy.mil/faq/docs/UT.html*，与*http://www.boulder.nist.gov/timefreq/general/misc.htm*，了解更多与时间标准有关的信息。

inode 变更时间是在文件建立之初，以及 inode metadata 被修改时被设置。

修改时间的变更是在文件块被更改，而非 metadata（文件名、用户、组、连接计数或权限）变更时。

`touch`命令，或`utime()`系统调用，可用于改变文件访问与修改时间，但不会改变inode变更时间。近期的 GNU `touch`版本提供选项，可针对文件标明时间。`ls -l` 命令显示的是修改时间，但加上 `-c` 选项，则可显示 inode 变更时间；加上 `-u` 选项，会显示访问时间。

这些时间戳都不够完美。inode 变更时间表示两种完全不同的目的，应该已经个别分开地被记录下来。因此，它并不能告诉你这个文件首度出现在 UNIX 文件系统里的确切时间。

访问时间在以`read()`系统调用读取文件时被更新，而不是在使用`mmap()`对应文件到内存以及以该方式读取文件时。

修改时间可能稍具可靠性，不过文件复制命令通常都会重设输出文件的修改时间为当前时间，即便它的内容完全没有变更，这并非我们所希望的，所以，复制命令 `cp` 提供 `-p` 选项，让你可以保留文件的修改时间。

最后备份的时间不会被记录：即备份系统必须保留补助性的数据，以追踪自最后一次备份至今已进行修改的文件。

注意： 文件系统备份软件，在保留文件时间戳这部分都相当谨慎处理。否则在每次备份之后，所有文件都看起来像刚被读取。使用打包工具，例如`tar`，作备份的系统，都必须更新inode变更时间，使得该时间戳无法再用于其他用途。

基于某些目的，有人希望能将读取、写入、更名、改变 metadata 的时间戳分开记录，这样的分隔方式在 UNIX 里是不可能的。

文件连接

尽管我们在本附录之前的“文件系统实现概况”讨论过，硬连接与软连接（符号连接）的工具非常多。但它们其实遭到一些非难，意见无非是同一个东西，给它多个名称只会混淆用户，因为连接是让两个已隔离的文件树接在一起。移动了含有连接的子树就会切断连接，让文件系统产生不一致的情况。图 B-4 展现的是因删除而切断了软连接的情况，而图 B-5 则是告诉你如何保留这样的连接，这完全是看你在建立连接时，是以相对还是绝对路径而定。

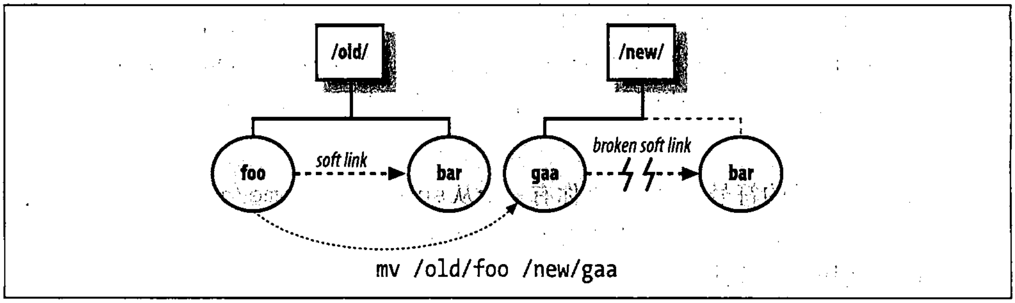


图 B-4：移动切断了软连接

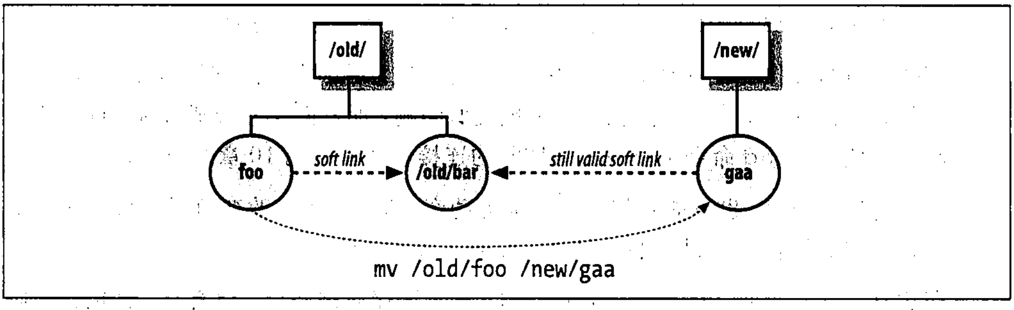


图 B-5：移动可以保留绝对符号连接

以下为硬连接与软连接会出现的其他问题：

- 当连接的文件更新时，不管它是被文件复制命令或是程序（例如，文本编辑程序）所替换，硬连接是否仍被保留根据更新的方式而定。如果是打开已存在的文件供输出及重写入，其 inode 编号保留不变，则硬连接仍会保留。然后，如果系统崩溃或磁盘溢满产生错误，则在更新期间可能导致遗失整个文件。比较小心的程序员可能就会在临时名称下编写新的版本，而且只有确定复制完成时，才删除原始的那个（因此连接计数减 1）并更改副本的名字。剩下的隐匿性很快，所以针对失误的窗口是较小的。替换文件会产生一个新的 inode 编号及连接计数 1，并切断硬连接。

  我们测试了许多文本编辑器，发现似乎都是使用第一种方式，保留硬连接。emacs 编辑器则允许在两种方式择一（注 22）。相对地，如果你编辑或重写的文件是软连接，那么你编修的就是原始数据，且只要它的路径名称仍未改变，则所有指向它的其他软连接，都会反映此更新过的内容。

注 22：将变量 backup-by-copying-when-linked 设为非 -nil (non-nil)，及 backup-by-copying 设为 nil，即可保留硬连接。可参考 emacs 手册里的 *Copying versus Renaming*。

以硬连接而言，两种更新方式都可能导致新文件的所有者与组改变：更新位置（update-in-place）会保留所有者与组，但复制与更名（copy-and-rename）则会将值重设为执行此操作的用户。因此，两种连接的行为模式在文件修改之后时常是不一致的。

- 再来看看目录的符号连接：如果你有一个从subdir到/home/jones/somedir的符号连接，那么当你将文件树移到另一个没有/home/jones/somedir的文件系统下时，连接便会被截断。
- 在连接里通常使用相对路径比较好，而且是只有在目录位于同级或更低级的情况下：所以从 subdir 到 ../anotherdir 的符号连接，只有在文件树至少比被移动的文件树高一层目录处开始才会被保留。否则，连接会被切断。
- 切断的符号连接无法在切断当时被发现，只有在之后你引用此连接时才会知道：这已经为时已晚。你的电话簿也可能出现类似问题：朋友搬了家没通知你，自此断了联系。使用 find 命令可以找出被切断的连接，请参考第 10 章的说明。
- 符号连接到目录，也可能对相对性目录更动产生问题：当你改变符号连接的父目录时，会移到被指向的目录的父目录，而非连接本身的父目录。
- 在建立文件打包时，符号连接会有问题：有时连接应被保留，但有时，打包文件应只是包括文件本身的副本而不是连接。

文件大小与时间戳的变化

每个文件包括的 inode 实体记录包含了它的字节大小，如果文件为空时它可以是零。ls 输出的冗长模式，将大小显示在第 5 栏：

```
$ ls -l /bin/ksh                                    列出冗长模式的文件信息
-rwxr-xr-x    1 root     root       172316 2001-06-24 21:12 /bin/ksh
```

GNU 版本的 ls 提供 -S 选项，以文件大小递减排序列出：

```
$ ls -lS /bin | head -n 8                          显示 8 个最大文件，并由大到小排列
total 7120
-rwxr-xr-x    1 rpm      rpm       1737960 2002-02-15 08:31 rpm
-rwxr-xr-x    1 root     root       519964 2001-07-09 06:56 bash
-rwxr-xr-x    1 root     root       472492 2001-06-24 20:08 ash.static
-rwxr-xr-x    2 root     root       404604 2001-07-30 12:46 zsh
-rwxr-xr-x    2 root     root       404604 2001-07-30 12:46 zsh-4.0.2
-rwxr-xr-x    1 root     root       387820 2002-01-28 04:10 vi
-rwxr-xr-x    1 root     root       288604 2001-06-24 21:45 tcsh
```

当文件系统使用空间已满，想要找出罪魁祸首时，-S 选项就派得上用场了。当然，如果你的 ls 不提供此选项，你也只要使用 ls -l *files* | sort -k5nr 便能得到相同的结果。

注意： 如果你怀疑某个正在执行的进程灌爆了文件系统，在Sun Solaris下，可以使用下列方式找到这个打开中的大文件（如果你想看到的不只是属于你的文件，请以root的身份执行）：

```
# ls -lS /proc/*/fd/*                                 列出所有打开的文件
-rw------- 1 jones jones 111679057 Jan 29 17:23 /proc/2965/fd/4
-r--r--r-- 1 smith smith  946643 Dec  2 03:25 /proc/15993/fd/16
-r--r--r-- 1 smith smith  835284 Dec  2 03:32 /proc/15993/fd/9
...
```

本例中，删除2965可能就能删除这个大文件 —— 至少你知道是jones引用它的了。

GNU/Linux也有/proc这样的工具机制，不过上面这个Solaris的解决方案在GNU/Linux下并不适用，因为它所报告的文件大小在GNU/Linux上是不正确的。

磁盘可用空间（disk-free）命令df用来报告当前磁盘的使用情况，或者你可加上-i选项了解inode的使用情况。磁盘使用情况命令du则可报告个别目录内容下的总使用空间，或辅以-s选项输出简洁的摘要。这些例子在第10章里都有。find命令搭配-mtime与-size选项，可以找出最近建立的或大小不寻常的文件，同样请参考第10章的说明。

在ls命令下使用-s选项可显示额外的开头栏位，其提供文件的块（block）大小：

```
$ ls -lgs /lib/lib* | head -n 4              以冗长模式列出前4个匹配文件的信息
2220 -r-xr-xr-t    1 sys  2270300 Nov  4  1999 /lib/libc.so.1
  60 -r--r--r--    1 sys    59348 Nov  4  1999 /lib/libcpr.so
 108 -r--r--r--    1 sys   107676 Nov  4  1999 /lib/libdisk.so
  28 -r--r--r--    1 sys    27832 Nov  4  1999 /lib/libmalloc.so
```

块大小与操作系统及文件系统息息相关：为了找到一个块的大小，可以字节为单位的文件大小除以用块为单位的文件大小，然后进制成2的次方，即可得知。以上述为例，我们发现2270300/2220 = 1022.6，所以其块大小为 $2^{10} = 1024$ 字节。随着存储设备的技术越来越精进，我们以块大小算出来的值可能与它所呈现在设备上的值有所不同。且厂商与某些GNU的ls版本也不一致，因此有时以此法取得的块大小不见得可靠－除非是在同系统下使用同一个ls命令作对照。

注意： 有时，你可能会遇到块小到有点奇怪的文件：像这样的文件多半有洞（hole），这是因为使用直接访问的方式写入字节在指定的位置。数据库程序就常这么做，因为它们是以松散式的表格存储在文件系统里。文件系统下的inode架构，处理有hole的文件时不会有问题，但对于读取这样文件的程序而言，它看到的可能是（想像的）磁盘块所对应至hole的一连串零字节。

复制如此的文件会以实体的零磁盘块填满hole，这可能会增加文件的大小。虽然建立原始文件的软件不会感觉到它，但它是提供功能齐备的备份工具所需要处理的一个文件系统功能。GNU的tar提供--sparse选项以请求检查这类文件，不过其他的tar实例则不提供。另外，GNU的cp也支持--sparse选项，以处理这类带有hole的文件。

使用管理的输出/恢复（dump/restore）工具，可能是在复制文件树时，唯一可避免填满hole的方法了。各系统里的此类工具都有很大的差异，所以我们在本书中不做讨论。

你可能还会发现在最后两个范例的输出上有个地方不同：时间戳的表示方式。为尽量缩减行宽度，`ls`通常是以*Mmm dd hh:mm*表示最近6个月内的时间戳，而以*Mmm dd yyyy*表示6个月前的时间。有些人会觉得这样很麻烦，而现行许多窗口系统都已经没有旧式ASCII终端那种80个字符的行限制了，因此这种做法已经不是那么必要。不过大部分的人仍认为太长的行会很难阅读，且近期的GNU `ls`版本也致力于将显示的结果保持在简短的样式。

GNU的`ls`会依locale的设置，显示近似*yyyy-mm-dd hh:mm:ss*这样的格式，以符合*ISO 8601:2000: Data elements and interchange formats-Information interchange-Representation of dates and times*的定义，不过就像先前的例子会去除秒数部分。

GNU的`ls`里，选项`--full-time`可用来揭露文件系统里完整的时间戳记录，如第10章所述。

其他的文件 metadata

剩下还有一些文件的属性记录在inode条目里，是我们还未提及的。不过在`ls -l`的输出里，还看到的部分就只有文件类型（file type）了，它记录在每行的第一个字符，就在权限的前面。`-`（连字号）指的是一般文件、`d`为目录，而`l`为符号连接。

这三种是我们在一般目录下常看到的，但在`/dev`下，你还会遇到至少这两种：`b`指块设备，`c`为字符设备。它们都与本书无关。

两种其他较少见的文件类型，例如`p`指的是命名的管道（named pipe），`s`指的是Socket（一种特殊的网络连接）。Socket为较高级的范畴，本书不作介绍。命名的管道则在程序与Shell脚本里偶尔用到：它们可以允许用户端和服务器端通过文件系统命名空间来沟通，并提供将一个进程的输出导向两个或两个以上不相关进程的方式。它们广义化一般的管道，后者只有一个写入与一个读取。

GNU *coreutils*包里的`stat`命令会显示`stat()`系统调用的结果，回传文件的inode信息。下面为SGI IRIX 里的使用范例：

```
$ stat /bin/true                                  报告文件的 inode 信息
  File: `/bin/true'
  Size: 312             Blocks: 8               IO Block: 65536  regular file
Device: eeh/238d        Inode: 380              Links: 1
Access: (0755/-rwxr-xr-x)  Uid: (    0/         root)   Gid: (    0/      sys)
Access: 2003-12-09 09:02:56.572619600 -0700
```

```
Modify: 1999-11-04 12:07:38.887783200 -0700
Change: 1999-11-04 12:07:38.888253600 -0700
```

这里显示的 stat 子集信息，比 ls 更细微。

GNU 的 stat 也支持设计更精细的报告，让你选择其中的子集输出。例如，软件安装包可使用它们，以找出文件系统是否仍有足够的空间可执行安装。详见 stat 手册页。

只有少数的UNIX版本（FreeBSD、GNU/Linux、NetBSD与SGI IRIX）支持原始的 stat 命令。这里举三个例子如下：

```
$ /usr/bin/stat /usr/bin/true   reeBSD 5.0 (较长的输出，已缩减长度以符合解说页面)
1027 1366263 -r-xr-xr-x 1 root wheel 5464488 3120 "Dec  2 18:48:36 2003"
"Jan 16 13:29:56 2003" "Apr  4 09:14:03 2003" 16384 8 /usr/bin/true

$ stat -t /bin/true               GNU/Linux 简洁的 inode 信息
/bin/true 312 8 81ed 0 0 ee 380 1 0 0 1070985776 941742458 941742458 65536

$ /sbin/stat /bin/true            SGI IRIX 系统工具程序
      /bin/true:
        inode 380; dev 238; links 1; size 312
        regular; mode is rwxr-xr-x; uid 0 (root); gid 0 (sys)
        projid 0        st_fstype: xfs
        change time - Thu Nov  4 12:07:38 1999 <941742458>
        access time - Tue Dec  9 09:02:56 2003 <1070985776>
        modify time - Thu Nov  4 12:07:38 1999 <941742458>
```

UNIX 文件的所有权与隐私权议题

我们已提及太多与文件权限相关的议题，让你了解如何控制文件与目录的读取、写入与执行的访问。你可以，也应该注意文件权限的选择，以掌控能访问你文件的有哪些人。

访问控制中最重要的工具就是umask命令了，因为它可以针对接下来建立的所有文件限制指定的权限。通常你会使用默认值，而它是设置在你Shell启动时所读取的文件里，以类似 sh 的 Shell 而言为 $HOME/.profile 文件，见 14.7 节。如 Shell 有支持，系统管理员通常会在对应的系统面起始文件内放置umask的设置。在协力合作的研究环境下，你应选择 022 掩码值，删除组与其他人的写入权限。以学生使用的环境来看，077 的掩码值较适合，可剔除所有除了所有者（与 root）以外的访问。

需要非默认权限时，Shell 脚本应于开始处明白地直接下达 umask 命令，这个操作必须在所有文件建立之前做。不过，像这样的设置不会影响在命令行上被重定向的那些文件，因为它们在脚本起始时，已被打开。

第二个重要的工具就是 chmod 命令了：你应该好好了解它。即便是在开放所有人读取的公开环境下，文件与目录仍应多作限制。包括邮件文件、网页浏览器历史记录与缓存、

私有信件、财务与个人数据、营销计划等。邮件客户端与浏览器通常使用的是默认的限制性权限，但你以文本编辑器建立的文件就必须自行使用chmod变更了。如果你想要更谨慎行事，那么请不要使用文本编辑程序建立文件：你可以先以touch建立空文件，执行 chmod 后再编辑它。

你应该还记得，系统管理员拥有你文件系统的完整访问权，他可以读取任何文件。虽然大部分的系统管理员认为未经文件所有者的允许，查看用户文件是不道德的，但部分组织却认为，所有计算机文件，包括电子邮件，都属于公司财产，应24小时监控。这点的合法性其实很模糊，且世界各国标准不一。

加密与数据安全性

如果你希望存储的文件除了你之外（几乎）没有任何人可以读取，那么就需要用到加密。由于各国政府的输出条例将加密工具视为武器，因此大部分UNIX厂商不会在它的标准发布包里随附加密软件。在你开始安装网络上找到的或是商用的加密软件之前，我们有以下几点建议：

- 安全性是一个程序，而非产品。有本书可以让你有更深入的了解：《Secrets and Lies: Digital Security in a Networked World》（Wiley）。
- 你是否曾忘记你的加密密钥，或是离职员工留下的密钥不正确，这都可能漏失你的数据：良好的加密方式，通常无法在你要的时间之内破解。
- 就像员工离职你可能会换门锁一样，你应该相信使用前职员的加密密钥是不可靠的，你应该使用新的密钥重新加密曾以先前的密钥加密过的所有文件。
- 如果加密文件提升安全性，使得用户很不方便，它们可能会直接停用加密。

如果你想了解更多与加密算法相关的历史，建议你从《The Code Book: The Evolution of Secrecy from Mary, Queen of Scots, to Quantum Cryptography》(Doubleday) 这本书开始。如果你觉得很有兴趣，想了解更多算法的细节，你可以继续看《Applied Cryptography: Protocols, Algorithms, and Source Code in C》(Wiley)。*http://www.math.utah.edu/pub/tex/bib/index-table.html* 里，还有更多相关学术议题的参考文献供你研究。

最后，在这个网络计算机的时代，你很可能会被网络与文件系统或是操作系统分开，除非你的网络通信确定相当安全，否则你的数据其实很危险。无线网络的弱点又更多，软件可以无声无息地窃听你的网络通信，利用现行无线加密通信协议的弱点，解出加密的通信数据。远端访问你的电子邮件，还有互动式的信息系统，可能都是不安全的。如果

你还在使用telnet，或非匿名式的ftp连接计算机，请立即切换为安全的Shell（secure Shell）（注 23）。旧式的这些通信软件会以明码文本传递所有数据，包括用户名称与密码；网络攻击者可以轻松地取得这类数据。*secure Shell* 软件使用健全的公钥加密，完成安全交换数据的操作，它会以随机产生长的加密密钥与其他许多较简单与较快的加密算法一起使用。

用户的数据会等到加密通道建立才开始传送，而标准的加密方法也经过深度的考虑，普遍相信是十分安全的。攻击者看到你的封包时，是经过随机字节组流加密后的样子，不过来源与目地端地址都看得到，也可能拿来分析。secure Shell也会为X Window System的数据建立安全通道，不过如果攻击者在你和你的计算机间动手脚，这么做也是于事无补的。网吧、键盘探测、无线网络等等，都可能让攻击大行其道，而令 secure Shell 无用武之地。

UNIX 扩展文件名惯例

有些操作系统，使用主要名称、一个点号，及1～3个字符的文件类型或文件扩展作为文件名的形式。这些扩展有其重要目的：指出文件内容属于哪种特定的数据类型。例如，扩展文件名为pas 指的是文件内容为 Pascal 的源代码，而 exe 指的则为二进制可执行文件。

这里并不保证文件扩展必会反映文件内容，不过大部分用户觉得这么做很好用，便遵循这一惯例。

UNIX也提供不少的通用文件扩展，但UNIX的文件名并未强制必须有一个点号。有时，文件扩展只是个惯例而已（对大部分的脚本语言来说），但编译器通常会要求特定的文件扩展及使用主文件名（截去扩展部分），以形成其他相关文件的名称。较常见的几种扩展见表 B-1。

表 B-1：常见的 UNIX 文件扩展

扩展	内容
1	数字的 1。手册页的 Section 1（用户命令）
a	程序库打包文件
awk	awk 语言原始文件

注 23： 例如*http://www.columbia.edu/kermit/*、*http://www.ssh.com/*与*http://www.openssh.org/*。要更深入了解 SSH，可参考《The Secure Shell: The Definitive Guide》（O'Reilly）一书。

表 B-1：常见的 UNIX 文件扩展（续）

扩展	内容
bz2	以 bzip2 压缩的文件
c	C 语言原始文件
cc C cpp cxx	C++ 语言原始文件
eps ps	PostScript 页面描述语言原始文件
f	Fortran 77 语言原始文件
gz	由 gzip 压缩的文件
f90	Fortran 90/95/200x 语言原始文件
h	C 语言标头文件
html htm	超文本标记语言（HyperText Markup Language）文件
o	目标文件（来自大部分编译程序语言）
pdf	可移植式文件格式文件
s	汇编语言源文件（例如，编译器的输出，以反映符号编码选项 -S）
sh	Bourne 系列 Shell 脚本
so	共享对象库（部分系统称之为动态载入库）
tar	磁带打包文件（产生自 tar 工具程序）
,v	cvs 与 rcs 的历史文件
z	以 pack 压缩的文件（极少见）
Z	以 compress 压缩的文件

此表格最明显的就是少了exe。虽然许多操作系统都使用它作为二进制可执行程序的扩展文件名，且允许在使用程序时，省略扩展文件名。但 UNIX 本身并未在可执行文件上应用任何特定的扩展（文件权限已经做了这件事了），UNIX的软件极少允许省略文件名扩展。

有些UNIX文本编辑程序提供用户建立临时备份文件，这么一来，就算是在编辑较久的通信期中，也能每隔一段时间就将文件记录在文件系统里。对这类备份文件的命名，使用惯例有几种：将（#）或（~）字符前置于文件开头或后置于文件结尾，或以~加上数字的方式编排，例如 .~1~、.~2~ 等等。后者的文件名产生编号方式模仿其他文件系统所提供的，UNIX本身并未提供这样的功能，不过其实UNIX的文件命名规则相当弹性，用户可以自行这么设置。

文件产生编号在其他系统下可保留文件的多个版本，而省略编号通常指的就是最高编号。UNIX 提供更好的方式，处理文件版本的历史记录：通过软件工具，保留与主文件不同

部分的历史记录，并附上注释性的描述，说明改变部分。这类的包一开始是AT&T的*Source Code Control System*（sccs），现今则为*Revision Control System*（rcs）（见附录C“其他程序”）与*Concurrent Versions System*（cvs）较常见。

小结

我们带你将UNIX的文件系统完整看过一遍，现在，你应该对下面这些功能已经相当熟悉了：

- 文件为0至多个8位字节的流，文本文件里的行只能以换行字符来标示行界限。
- 字节通常以ASCII字符解释，但UTF-8编号与Unicode字符集让UNIX文件系统、管道与网络通信，能支持全世界书写世界上数百万种不同字符，且大部分现存文件或软件也能有效使用。
- 文件有属性，例如时间戳、所有权与权限。这可以让我们更有效地设置访问控制层级及隐私权，提供比其他桌面环境操作系统更好的支持，并剔除大部分计算机病毒的问题。
- 可以在目录的单一节点设置适当权限，控制整个目录树的访问。
- 极大的文件几乎不会造成什么问题，而在现行技术下，新的文件系统设计也能容许越来越大的文件。
- 文件名与路径名称的最大长度，已超出你会用到的最大长度。
- 简明的层级式目录架构，辅以斜杠分隔的路径组成部分，搭配`mount`命令的使用，几乎可以拥有无限大小的逻辑式文件系统。
- 尽可能地将所有数据看成文件，且鼓励这么做，可以简化人为的数据处理与使用。
- 文件名可使用NUL与斜杠以外的所有字符，但实务上，为了可移植性、可靠度，及Shell通配字符的考虑，大大地限制了应该可被使用的字符。
- 文件名的字母大小写将视为不同（除了Mac OS X的非UNIX HFS文件系统）。
- 虽然文件系统本身并未规定文件名结构，但许多程序仍预期文件名应有.加上扩展名称，并利用此扩展建立相关文件。Shell通过它们对通配字符样式的支持，如`ch01.*`与`*.xml`，来鼓励此实现。
- 文件名存储在目录文件里，而与文件相关的信息、文件*metadata*则另存储于inode实体记录中。
- 在相同文件系统内的文件与目录要移动或更名是相当快的，因为只有它们包含的目录实体需被更新，文件数据块本身则不会被访问。

- 硬连接与软连接允许同一实体文件拥有多个名称。硬连接只限在单一实体文件系统里才能做，但软连接则可指向逻辑文件系统的任一位置。
- inode表格大小为文件系统安装时就已固定，所以文件系统即使仍有空间置放文件数据，也可能出现已满的状态。

附录 C

重要的 UNIX 命令

现今 UNIX 系统都随附相当多命令。很多是特殊用途，也有很多是日常处理使用的，它们都能应用在交互模式下，或写在 Shell 脚本里。不过，我们不可能包括得了系统里所有的命令，也没有这么做的必要（像《UNIX in a Nutshell》的书籍对此部分有非常深入的描述）。

我们仍尽可能地找出有用的命令，也就是UNIX的用户或程序设计人员首先应了解的那些，简单做个介绍。这里也可能包括早期UNIX使用的旧式命令。本附录只是当你有志于成为 UNIX 的开发者时，建议你研究的命令列表。为求简洁，我们重新做了分类，用表这些命令列表，并进行简单的说明。

Shell 与内置命令

首先，我们先了解 Bourne Shell 的部分，特别是 POSIX 整理过的那些。bash 与 ksh93 都为 POSIX 兼容，而另外还有一些 Shell 在语法上也与 Bourne Shell 一致：

bash	GNU Project 的 Bourne-Again Shell。
ksh	Korn Shell —— 原始版本或分支体系版本，视操作系统而定。
pdksh	Public Domain Korn Shell。
sh	原始 Bourne Shell，特别是在商用的 UNIX 系统上。
zsh	Z-Shell。

你应该了解 Shell 内置命令的运作方式：

.	在当前 Shell 下，读取与执行给定的文件。
break	切断 for、select、until 或 while 循环。
cd	更改当前的目录。

command	规避函数的查找，直接执行正规的内置命令。
continue	开始 for、select、until，或while 循环的下一个重复。
eval	将给定的文本视为 Shell 命令。
exec	无参数的情况下，改变 Shell 打开的文件。如带有参数，则以其他程序置换 Shell。
exit	退出 Shell 脚本，可选地带有特定的退出码。
export	将变量导出到接下来的程序环境中。
false	什么事也不做，指非成功的状态。用于 Shell 循环中。
getopts	处理命令行选项。
read	将输入行读进一个或多个 Shell 变量里。
readonly	将变量标记为只读，例：不可更改的。
return	返回自 Shell 函数而来的值。
set	显示Shell变量与变量值、设置Shell选项、设置命令行参数（$1、$2、…）。
shift	一次移动一个或多个命令行参数。
test	计算表达式，检测其为字符串、数字或文件属性相关的。
trap	管理操作系统信号。
true	什么事也不做，指成功的状态。用于 Shell 循环中。
type	指出命令的特性（关键字、内置命令、外部命令等等）。
typeset	声明变量与管理它们的类型与属性。
ulimit	设置或显示系统对每个进程所加诸的限制。
unset	删除 Shell 变量与函数。

下列为编写日常处理的 Shell 脚本的好用命令：

basename	显示路径名称的最后元件，并可选用地删除副文件名。主要用于命令替换。
dirname	显示除了路径名称最后组成部分以外的所有信息。主要用于命令替换。
env	处理命令的环境。
id	显示用户与组 ID 及名称信息。
date	显示现在的日期与时间，可选用地受用户提供的格式字符串所控制。
who	显示已登录的用户列表。
stty	处理当前终端设备的状态。

文本处理

下面的命令是做文本处理用。

awk	优雅又实用的程序语言，为许多大型 Shell 脚本的重要组成部分。
cat	连接文件。
cmp	简单的文件比较程序。

cut	剪下选定的列或字段。
dd	阻绝与解除阻绝数据的专门程序，也可执行ASCII与EBCDIC之间的转换。dd在产生设备文件原貌的副本时特别好用。需要特别注意的是，执行字符集转换时使用iconv较为合适。
echo	将参数打印到标准输出。
egrep	扩展的grep。使用扩展正则表达式(Extended Regular Expressions；ERE)进行匹配。
expand	展开制表字符与空格字符。
fgrep	快速grep。此程序使用与grep不同的算法匹配固定字符串。大部分的UNIX系统可同时查找多个固定字符串——不过并非全部。
fmt	将文本格式化为段落的简单工具。
grep	源自原始的ed行编辑命令g/*re*/p，"全局性匹配正则表达式并打印"。使用基本正则表达式（Basic Regular Expressions，BREs）进行匹配。
iconv	一般用途的字符编码转换工具。
join	自多个文件结合匹配的记录。
less	设计精良的交互式分页（pager）程序用以于终端上查看信息，一次显示屏幕所能显示的内容（页）。现已有GNU Project提供此程序，其名称为对应的more程序双关语。
more	原始的BSD UNIX交互式分页程序。
pr	将文件格式化，供行打打印机使用。
printf	echo的精装版，提供要被打印参数的控制方式。
sed	流编辑器，以ed行编辑器的命令集为基础。
sort	排序文本文件。命令行参数提供排序键值的指定与优先级控制。
spell	批次拼字检查程序。你也可以使用aspell或ispell封装成名为spell的Shell脚本。
tee	将标准输入拷贝到标准输出，或到一至多个指名的输出文件。
tr	转换、删除或减少重复字符的执行。
unexpand	将空格字符转换成适当数量的制表字符。
uniq	删除或计算已排序输入中的重复行。
wc	计算行、单词、字符或字节。

文件

与文件处理相关的命令：

bzip2, bunzip2	极高品质的文件压缩与解压缩。
chgrp	更改文件与目录的组。
chmod	更改文件与目录的权限（模式）。

chown	更改文件或目录的所有权。
cksum	显示文件的校验和（checksum）、POSIX 标准算法。
comm	显示或省略两个排序后的文件之间具有唯一性或共有的行。
cp	复制文件与目录。
df	显示可用磁盘空间。
diff	比较文件，显示其差异。
du	显示文件与目录所使用的磁盘块。
file	通过文件开头部分的检查，判断文件里的数据类型。
find	向下一个或多个目录阶层，寻找匹配于指定条件的文件系统对象（文件、目录、特殊文件）。
gzip, gunzip	高品质的文件压缩与解压缩。
head	显示一个或多个文件的前 *n* 行。
locate	以文件名称在系统里查找一文件。此程序使用定期自动重建的文件数据库中进行查找。
ls	列出文件。可使用选项控制要显示的信息。
md5sum	打印文件校验和，其使用 Message Digest 5（MD5）算法求出校验和。
mktemp	建立独一无二的临时文件，并显示其名称。非所有系统都可使用。
od	八进制输出；以八进制、十六进制或作为字符数据来打印文件内容。
patch	通过读取 diff 的输出，将给定的文件更新为新版本。
pwd	显示当前的工作目录。通常内置在现代的 Shell 中。
rm	删除文件与目录。
rmdir	只删除空目录。
strings	查找二进制文件中可打印的字符串，并显示它们。
tail	显示文件的最后 *n* 行。加上 -f 则继续打印（成长）文件的内容。
tar	磁带打包程序。现常被拿来作为软件发布的格式。
touch	更新文件的修改或访问时间。
umask	设置默认的文件建立权限掩码。
zip, unzip	文件打包与压缩/解压缩程序。ZIP 格式可使用于多种操作系统下，相当具有可移植性。

进程

以下为建立、删除，或管理进程所使用的命令：

at	在指定时间执行工作。at 调度的工作只执行一次，而 cron 则为定期执行。
batch	在系统负载较不忙碌时，执行工作。
cron	在指定时间执行工作。

crontab	编辑每个用户的“cron 表格”文件，指定应执行哪些命令，于何时执行。
fuser	寻找正在使用特定文件或 socket 的进程。
kill	传送信号到一或多个进程。
nice	在进程执行前，更改其优先级。
ps	进程状态。显示与正在执行中进程有关的信息。
renice	进程已被启动后，更改其优先级。
sleep	停止执行一段指定的秒数。
top	交互式显示系统上密集使用 CPU 的工作。
wait	Shell 内置命令，等待一个或多个进程完成。
xargs	读取标准输入上的字符串，作为参数，尽可能地传递给指定的命令。多半会搭配 find 使用。

其他程序

还有些其他范畴的命令：

cvs	Concurrent Versions System，功能强大的源代码管理程序。
info	GNU Info 系统，供在线文件浏览使用。
locale	显示可用的 locale 相关信息。
logger	通常是通过 *syslog*(3)，传送信息到系统日志文件。
lp, lpr	将打印缓冲区文件传送给打印机。
lpq	显示正在处理中与在队列等待中的打印工作列表。
mail	传送电子邮件。
make	控制文件的编译与重新编译。
man	显示命令、程序库函数、系统调用、设备、文件格式与管理性命令的在线手册页。
scp	安全进行文件的远端复制。
ssh	安全的 Shell。在执进程序或交互式登录的机器之间提供加密的连接。
uptime	显示系统已开机多久及其负载信息。

在其他种类中，针对 Revision Control System（RCS）的相关命令如下：

ci	签入文件到 RCS。
co	自 RCS 中签出文件。
rcs	在 RCS 控制下处理文件。
rcsdiff	对 RCS 控制下的文件的两个不同版本，执行 diff。
rlog	为一至多个 RCS 所管理的文件，打印签入（check-in）日志。

参考书目

UNIX 程序员的手册

1. 《UNIX Time-sharing System: UNIX Programmers Manual》, Seventh Edition, Volumes 1, 2A, 2B. Bell Telephone Laboratories, Inc., January 1979.

   这些是第7版（Seventh Edition）UNIX系统的参考手册（Volume 1）及描述性报告（Volumes 2A & 2B），UNIX 系统第7版是所有当前商用UNIX系统的直属祖先。

   它们由 Holt Rinehart & Winston 出版，但是现在已经很久未再印刷了。然而，它们在 Bell Labs 网站有在线形式，采用的格式有 `troff`、PDF 及 PostScript。参考 *http://plan9.bell-labs.com/7thEdMan*。

2. 你的 UNIX 程序设计手册。最有启发性的指导书之一，你可以从头读到尾（注 1）（当UNIX系统已经长大，这会比以前更为困难）。如果你的UNIX厂商将它的文件印刷出书，则会较容易些。否则，需要从 Seventh Edition 手册开始，然后也需要阅读你的本地端文件。

使用 UNIX 思路进行程序设计

我们期待此书已经帮助你用现代文字学习“思考UNIX”。此列表的前两本书是UNIX“工具箱”程序设计方法论的原始展现。第3本书检查在 UNIX 下可用的、更广的程序设计

注 1：在我作为合同程序员的某个夏日，我用几次午餐的时间从头至尾地阅读了这本手册。没想到我在那么短的时间里学到这么多知识。

工具。第4本与第5本是一般性的程序设计图书，且非常值得阅读。我们注意到由Brian Kernighan编写的任何书籍都值得仔细阅读，通常是要读许多遍。

1. 《Software Tools》，Brian W. Kernighan and P. J. Plauger. Addison-Wesley, Reading, MA, U.S.A., 1976. ISBN 0-201-03669-X.

   一本展现相当于UNIX grep、sort、ed等程序设计与源代码的很棒书籍（注2）。其程序使用Ratfor（Rational Fortran），是具有仿C控制结构的Fortran预处理器。

2. 《Software Tools in Pascal》，Brian W. Kernighan and P. J. Plauger. Addison-Wesley, Reading, MA, U.S.A., 1981. ISBN 0-201-10342-7.

   前一本书转成Pascal版。仍然值得阅读，Pascal提供了许多Fortran没有的事情。

3. 《The UNIX Programming Environment》，Brian W. Kernighan and Rob Pike. Prentice-Hall, Englewood Cliffs, NJ, U.S.A., 1984. ISBN 0-13-937699-2 (hardcover), 0-13-937681-X (paperback).

   本书的焦点在UNIX，并在该环境中使用工具。特别地，它增加重要的题材在Shell、awk及lex的使用上。参考 *http://cm.bell-labs.com/cm/cs/upe*。

4. 《The Elements of Programming Style》，Second Edition, Brian W. Kernighan and P. J. Plauger. McGraw-Hill, New York, NY, U.S.A., 1978. ISBN 0-07-034207-5.

   Strunk & White有名的《The Elements of Style》之后的经典书籍，本书描述可以用在任何环境的优良程序设计实务。

5. 《The Practice of Programming》，Brian W. Kernighan and Rob Pike. Addison-Wesley Longman, Reading, MA, U.S.A., 1999. ISBN 0-201-61586-X.

   与前一部书相似，但具有更强的技术性。参考 *http://cm.bell-labs.com/cm/cs/tpop*。

6. 《The Art of UNIX Programming》，Eric S. Raymond. Addison-Wesley, Reading, MA, U.S.A., 2003. ISBN 0-13-124085-4.

7. 《Programming Pearls》，First Edition, Jon Louis Bentley. Addison-Wesley, Reading, MA, U.S.A., 1986. ISBN 0-201-10331-1.

8. 《Programming Pearls》，Second Edition, Jon Louis Bentley. Addison-Wesley, Reading, MA, U.S.A., 2000. ISBN 0-201-65788-0.

   参考 *http://www.cs.bell-labs.com/cm/cs/pearls/*。

9. 《More Programming Pearls: Confessions of a Coder》，Jon Louis Bentley. Addison-Wesley, Reading, MA, U.S.A., 1988. ISBN 0-201-11889-0.

注2：　这本书长久地改变了我的生活。

Bentley 编写的象征 UNIX 精神的优秀书籍，且具有极少语言、算法设计，与更多的美妙范例。这些应该是在每个正规程序员的书架上。

10. 《Linux and the UNIX Philosophy》, Mike Gancarz. Digital Press, Bedford, MA, U.S.A., 2003. ISBN 1-55558-273-7.

Awk and Shell

1. 《The AWK Programming Language》, Alfred V. Aho, Brian W. Kernighan, and Peter J. Weinberger. Addison-Wesley, Reading, MA, U.S.A., 1987. ISBN 0-201-07981-X.

   awk 程序语言的原始定义。相当值得阅读。参考 ***http://cm.bell-labs.com/cm/cs/awkbook***。

   《Effective awk Programming》, Third Edition, Arnold Robbins. O'Reilly, Sebastopol, CA, U.S.A., 2001. ISBN 0-596-00070-7.

   awk 的指导手册，涵盖 POSIX 标准的 awk。它也可作为 gawk 的用户指引。

2. 《The New KornShell Command and Programming Language》, Morris I. Bolsky and David G. Korn. Prentice-Hall, Englewood Cliffs, NJ, U.S.A., 1995. ISBN 0-13-182700-6.

   在 Korn Shell 上最可靠的作品。

3. 《Hands-On KornShell93 Programming》, Barry Rosenberg. Addison-Wesley Longman, Reading, MA, U.S.A., 1998. ISBN 0-201-31018-X.

标准

正式的标准文件是重要的，因为它们表示实现者与计算机系统用户之间的“合约”。

1. 《IEEE Standard 1003.1-2001: Standard for Information Technology-Portable Operating System Interface》(POSIX®) IEEE, New York, NY, U.S.A., 2001.

   这是次新的 POSIX 标准。它结合系统调用界面与 Shell 及工具标准在一份文件中。实际上此标准包括许多册，可在线取得（注 3），通过 PDF 电子格式打印（注 4），及由 CD-ROM 提供：

注 3： 参考网址：***http://www.opengroup.org/online pubs/007904975***。

注 4： 参考网址：***http://www.standards.ieee.org/***。

基本定义

此册提供标准的历史、术语的定义及文件格式与输出入格式的规范。ISBN 0-7381-3047-8；PDF: 0-7381-3010-9/SS94956；CD-ROM: 0-7381-3129-6/SE94956。

基本原理（信息型）

不是标准的正式部分，它不会在实例上加诸需求，此册提供 POSIX 标准中为何事情是如此的说明。ISBN 0-7381-3048-6；PDF: 0-7381-3010-9/SS94956；CD-ROM: 0-7381-3129-6/SE94956。

系统界面

描述 C 或 C++ 程序员能看到操作系统的界面。ISBN 0-7381-3094-4；PDF: 0-7381-3010-9/SS94956；CD-ROM: 0-7381-3129-6/SE94956。

Shell 与工具

对于本书的读者更为相关：它描述在 Shell 与工具层次的操作系统。ISBN 0-7381-3050-8；PDF: 0-7381-3010-9/SS94956；CD-ROM: 0-7381-3129-6/SE9。

2. 《IEEE Standard 1003.1-2004: Standard for Information Technology-Portable Operating System Interface》(POSIX®) IEEE, New York, NY, U.S.A., 2004.

最新的 POSIX 标准，当本书付印时发行。它是前一版的修订版，且以相似的方式组织。此标准包括许多册：基本定义（Volume 1）、系统界面（Volume 2）、Shell 与工具（Volume 3）和基本原理（Volume 4）。

此标准可从 *http://www.standards.ieee.org/* 订购 CD-ROM（产品编号 SE95238，ISBN 0-7381-4049-X）或是 PDF（产品编号 SS95238，ISBN 0-7381-4048-1）。

3. 《The Unicode Standard》, Version 4.0, The Unicode Consortium. Addison-Wesley, Reading, MA, U.S.A., 2003. ISBN 0-321-18578-1.

4. XML 的标准，可从 *http://www.w3.org/TR/REC-xml/* 中取得。

安全与加密

1. 《PGP: Pretty Good Privacy》, Simson Garfinkel. O'Reilly, Sebastopol, CA, U.S.A., 1995. ISBN 1-56592-098-8.

2. 《The Official PGP User's Guide》, Philip R. Zimmermann. MIT Press, Cambridge, MA, U.S.A., 1995. ISBN 0-262-74017-6.

3. 《Practical UNIX & Internet Security》, Third Edition, Simson Garfinkel, Gene

Spafford, and Alan Schwartz. O'Reilly, Sebastopol, CA, U.S.A., 2003. ISBN 0-596-00323-4.

4. 《SSH, The Secure Shell: The Definitive Guide》, Second Edition, Daniel J. Barrett, Richard E. Silverman, and Robert G. Byrnes. O'Reilly Media, Sebastopol, CA, U.S.A., 2005. ISBN 0-596-00895-3.

5. 《Secrets and Lies: Digital Security in a Networked World》, Bruce Schneier. Wiley, New York, NY, U.S.A., 2000. ISBN 0-471-25311-1.

   本书深刻地揭露计算机安全性对每一个世界公民在生活上、数据上及个人自由上的含意。Bruce Schneier 如同 Brian Kernighan、Jon Bentley 与 Donald Knuth 一样，总是值得阅读的作家之一。

6. 《The Code Book: The Evolution of Secrecy from Mary, Queen of Scots, to Quantum Cryptography》, Simon Singh. Doubleday, New York, NY, U.S.A., 1999. ISBN 0-385-49531-5.

7. 《Applied Cryptography: Protocols, Algorithms, and Source Code in C》, Second Edition, Bruce Schneier. Wiley, New York, NY, U.S.A., 1996. ISBN 0-471-12845-7 (hardcover), 0-471-11709-9 (paperback).

8. 《Cryptographic Security Architecture: Design and Verification》, Peter Gutmann. Springer-Verlag, New York, NY, U.S.A., 2004. ISBN 0-387-95387-6.

UNIX 内部

1. 《Lions' Commentary on UNIX 6th Edition, with Source Code》, John Lions. Peer-to-Peer Communications, 1996. ISBN 1-57398-013-7.

2. 《The Design and Implementation of the 4.4BSD Operating System》, Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman. Addison-Wesley, Reading, MA, U.S.A., 1996. ISBN 0-201-54979-4.

3. 《UNIX Internals: The New Frontiers》, Uresh Vahalia. Prentice Hall, Englewood Cliffs, NJ, U.S.A., 1996. ISBN 0-13-101908-2.

O'Reilly 书籍

这里是 O'Reilly 书籍的列表。当然，还有许多与 UNIX 相关的 O'Reilly 书籍，可参考 *http://www.oreilly.com/catalog*。

1. 《Learning the bash Shell》, Third Edition, Cameron Newham and Bill Rosenblatt. O'Reilly, Sebastopol, CA, U.S.A., 2005. ISBN 0-596-00965-8.
2. 《Learning the Korn Shell》, Second Edition, Bill Rosenblatt and Arnold Robbins. O'Reilly, Sebastopol, CA, U.S.A., 2002. ISBN 0-596-00195-9.
3. 《Learning the UNIX Operating System》, Fifth Edition, Jerry Peek, Grace Todino, and John Strang. O'Reilly, Sebastopol, CA, U.S.A., 2001. ISBN 0-596-00261-0.
4. 《Linux in a NutShell》, Third Edition, Ellen Siever, Stephen Spainhour, Jessica P. Hekman, and Stephen Figgins. O'Reilly, Sebastopol, CA, U.S.A., 2000. ISBN 0-596-00025-1.
5. 《Mastering Regular Expressions》, Second Edition, Jeffrey E. F. Friedl. O'Reilly, Sebastopol, CA, U.S.A., 2002. ISBN 0-596-00289-0.
6. 《Managing Projects with GNU make》, Third Edition, Robert Mecklenburg, Andy Oram, and Steve Talbott. O'Reilly Media, Sebastopol, CA, U.S.A., 2005. ISBN: 0-596-00610-1.
7. 《sed and awk》, Second Edition, Dale Dougherty and Arnold Robbins. O'Reilly, Sebastopol, CA, U.S.A., 1997. ISBN 1-56592-225-5.
8. 《sed and awk Pocket Reference》, Second Edition, Arnold Robbins. O'Reilly, Sebastopol, CA, U.S.A., 2002. ISBN 0-596-00352-8.
9. 《UNIX in a NutShell》, Third Edition, Arnold Robbins. O'Reilly, Sebastopol, CA, U.S.A., 1999. ISBN 1-56592-427-4.

其他书籍

1. 《CUPS: Common UNIX Printing System》, Michael R. Sweet. SAMS Publishing, Indianapolis, IN, U.S.A., 2001. ISBN 0-672-32196-3.
2. 《SQL in a NutShell》, Kevin Kline and Daniel Kline. O'Reilly, Sebastopol, CA, U.S.A., 2000. ISBN 1-56592-744-3.
3. 《HTML & XHTML: The Definitive Guide》, Chuck Musciano and Bill Kennedy. O'Reilly, Sebastopol, CA, U.S.A., 2002. ISBN 0-596-00026-X.
4. 《The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary》, Eric S. Raymond. O'Reilly, Sebastopol, CA, U.S.A., 2001. ISBN 0-596-00131-2 (hardcover), 0-596-00108-8 (paperback).

5. 《Texinfo: The GNU Documentation Format》, Robert J. Chassell and Richard M. Stallman. Free Software Foundation, Cambridge, MA, U.S.A., 1999. ISBN 1-882114-67-1.

6. 《The TEXbook》, Donald E. Knuth. Addison-Wesley, Reading, MA, U.S.A., 1984. ISBN 0-201-13448-9.

7. 《The Art of Computer Programming, Volume 2: Seminumerical Algorithms》, Third Edition, Donald E. Knuth. Addison-Wesley, Reading, MA, U.S.A., 1997. ISBN 0-201-89684-2.

8. 《Literate Programming》, Donald E. Knuth. Stanford University Center for the Study of Language and Information, Stanford, CA, U.S.A., 1992. ISBN 0-937073-80-6 (paperback) and 0-937073-81-4 (hardcover).

9. 《Herman Hollerith —Forgotten Giant of Information Processing》, Geoffrey D. Austrian. Columbia University Press, New York, NY, U.S.A. 1982. ISBN 0-231-05146-8.

10. 《Father Son & Co. — My Life at IBM and Beyond》, Thomas J. Watson Jr. and Peter Petre. Bantam Books, New York, NY, U.S.A., 1990. ISBN 0-553-07011-8.

11. 《A Quarter Century of UNIX》, Peter H. Salus. Addison-Wesley, Reading, MA, U.S.A., 1994. ISBN 0-201-54777-5.

作者简介

Arnold Robbins是亚特兰大人，他是一位专业程序员与技术作家。他也是一位快乐的丈夫，4个可爱孩子的父亲，还是犹太教徒（Babylonian 与 Jerusalem）。1997 年末，他与家人移居到以色列。

Arnold 从 1980 年开始使用 UNIX 系统，那是一台执行第 6 版 UNIX（Sixth Edition UNIX）的PDP-11。从1984年开始他已经着手Shell脚本程序编写，先从增强型Bourne Shell 开始，然后使用 Korn Shell 与 bash。

自 1987 年开始，Arnold 也已经是重度的 awk 用户，当时他使用的是 gawk —— awk 的 GNU 版。身为 POSIX 1003.2 的委员，他曾协助 awk 的 POSIX 标准建立。他目前是 gawk 及其文件的维护者。

他曾是个系统管理者以及UNIX与网络持续教育课程的老师。他也曾拥有一家创业型软件公司，但是他已不想再提起。他希望有一天能将他自己的网站放在 *http://www.skeeve.com*。

O'Reilly 已经使他很忙了。他是畅销书《Learning the vi Editor》、《Effective awk Programming》、《sed and awk》、《Learning the Korn Shell》、《UNIX in a Nutshell》以及许多口袋参考书的合著者。

Nelson H. F. Beebe 是犹他大学数学系研究所教授，具有化学、物理、数学、电子计算机科学以及计算工具管理等后台。他曾在几个主要制造商的计算机上工作过许多年。他总是会在个人计算机中为每种UNIX打分。他是许多程序语言（包括awk）、浮点算术、软件可移植性、科学软件与计算机图形的专家，而且长期从事在早期UNIX时代的电子文件与排版。

封面介绍

我们的外观是取自于读者建议、我们自己的经验及经销商的回馈。与众不同的封面结合我们对技术主题特有的表现方式，将个性与生活融入较为艰涩的主题。

本书的动物封面是非洲的帐蓬陆龟(Psammobates tentorius)，其龟甲呈现几何形状。Psammobates的含义是“热爱沙地”，其栖息地包括干燥的草原、沙漠边缘及沿海沙地，属于典型的干燥炎热气候。所以帐蓬陆龟只能在南非的草原与沙漠外围找到，这一点也不令人惊讶。这种类型的所有品种都是小型的，大小约从5英寸到10英寸，而且在它们的甲壳上有黄色放射型的标记。帐蓬陆龟特别突出的特征是具有拱起的龟甲。

陆龟以它们的长寿出名，而且乌龟与陆龟也是今日最古老的物种。它们在2亿年前的恐龙时代就已存在。所有的龟类都依赖温度，也就是它们只在温度不是太极端时才吃东西。在酷暑与寒冬期间，陆龟会冬眠且群体停止进食。在春天，陆龟的食物包括多汁植物、纤维植物及牧草。

计算机精品学习资料大放送